Eidgenössische
Technische Hochschule
Zürich

Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Distributed Systems Group
Prof. Friedemann Mattern

Inspektion von Sensornetzen per PDA

Semesterarbeit im SS05

Bearbeitet von:

Mustafa Yücel

Betreuer:

Prof. Friedemann Mattern, Matthias Ringwald

ETH Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Distributed Systems Group
Prof. Friedemann Mattern

Semesterarbeit SS05

Inspektion von Sensornetzen per PDA

Der an der ETH entwickelte BTnode (Mikrocontroller, Bluetooth-Modul und zusätzliches Low-power-Funkmodul) kann zur Realisierung von Sensornetzen genutzt werden. Dabei agieren die einzelnen Knoten autonom und kommunizieren drahtlos miteinander. Während der Entwicklung und auch später im Betrieb ist es hilfreich, wenn man in solchen Sensornetz-Anwendungen vor Ort einzelne Knoten inspizieren kann. Beispielsweise könnte der Zustand des Akkus abgefragt werden oder die Topologie aller verbundenen Knoten angezeigt werden.

In dieser Semesterarbeit soll ein PDA zur Inspektion von BTnode-Netzen verwendet werden, um über die Bluetooth-Schnittstelle mit einzelnen BTnodes zu kommunizieren und verschiedene Daten auf dem Handheld darzustellen. Hierzu ist es erforderlich, eine einfache, aber generische Schnittstelle zur Bereitstellung von Informationen auf dem BTnode zu entwickeln, um dann verschiedene Basisdienste anbieten zu können. Einen wesentlichen Aspekt der Arbeit nimmt ausserdem die Topologiedarstellung ein.

Als Betriebssystem wird auf den BTnodes das freie Ethernut zusammen mit einem, in einer früheren Studentenarbeit entstandenen, Bluetooth-Stack verwendet. Als PDA ist ein iPAQ unter Linux vorgesehen und die Entwicklung des Diagnosewerkzeuges kann in J2SE mit JSR-82 erfolgen.

Voraussetzungen: Solide C- und Java-Kenntnisse, Grundkenntnisse über Betriebssysteme (Threads, Events, Mutexe, ...).

Bearbeitet von: Mustafa Yücel
Betreuer: Prof. Friedemann Mattern, Matthias Ringwald

Zusammenfassung

Ein PDA wird für die Wartung und Inspektion von Sensornetzen eingesetzt. Ein Sensornetz ist ein Rechnernetz aus Kleinst-Computern, sogenannten Sensorknoten, die mit Sensoren ausgestattet sind und durch Zusammenarbeit eine gemeinsame Aufgabe bewältigen. Der BTnode ist ein solcher Sensorknoten. Für die drahtlose Kommunikation wird Bluetooth genutzt. Bluetooth eignet sich aufgrund des niedrigen Stromverbrauchs für mobile Kleingeräte. Der PDA ist somit eine handliche Lösung, um mit den BTnodes zu kommunizieren. Es wird auf einer plattform-unabhängigen Software-Umgebung programmiert: Java bietet eine Standard-Bluetooth-API unter der Spezifikation JSR-82 an. Ursprünglich für Mobiltelefone entwickelt, sind auch einige Implementierungen für die Java Standard Edition (J2SE) verfügbar. Als Betriebssystem wird Linux benutzt, da auf dieser Plattform bereits freie JSR-82-Implementierungen existieren und unter anderem für die Entwicklung hilfreiche Softwarepakete mitgeliefert werden.
Für den PDA wurden drei Applikationen programmiert. Die erste Applikation (RemoteProgramming) ermöglicht die Neuprogrammierung der BTnodes über Bluetooth. Die zweite Applikation (AttributeRetrieving) greift auf eine eigens programmierte, generische Architektur zurück, um Informationen von einem BTnode abzufragen. In der dritten Applikation (TopologyDrawing) läuft auf dem BTnode eine Ad-Hoc-Netz-Software namens JAWS. Die Applikation stellt die Topologie des Netzes grafisch als Baum auf dem PDA dar.

Vorwort

Im Sommersemester 2005 habe ich die Möglichkeit wahrgenommen, - ausserhalb des Departements für Elektrotechnik und Informationstechnologie - am Informatik-Departement eine Semesterarbeit zu schreiben. Die Vertiefung in Java, der Einblick in die Hardware und Software der BTnodes, der Einsatz von Linux auf dem PDA sowie die Bluetooth-Kommunikation waren für mich interessante Sachgebiete, welche mich zu dieser Semesterarbeit motivierten.
Dank den breiten Erfahrungen meines Betreuers Matthias Ringwald war es mir möglich, mich in das Gebiet Informatik zu vertiefen. Ich möchte mich für die Unterstützung von Matthias, welcher mir bei Problemen kompetent zur Seite stand, bedanken.

Zürich, Herbst 2005

Mustafa Yücel

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Dieses Kapitel dient als Einführung in die Grundlagen dieser Arbeit. Es beschreibt einerseits die Komponenten des BTnode und andererseits wird auf die Hardware und Software des PDA näher eingegangen. Dabei dient Bluetooth als Übertragungskanal zwischen beiden Geräten. Als Betriebssystem für den PDA wird eine Linux-Distribution verwendet. In diesem Zusammenhang wird eine Toolchain genutzt, mit der man Programme für den PDA bzw. dessen Prozessortyp kompilieren kann. Als Programmiersprache wird auf dem PDA Java verwendet. Für die Bluetooth-Kommunikation und für die grafische Oberfläche existieren standardisierte Schnittstellen und die dazugehörigen Bibliotheken. Damit sind die Applikationen von der verwendeten Plattform weitgehend unabhängig, solange eine Java-Laufzeitumgebung und eine Implementierung der Bluetooth-Schnittstelle für Java existiert.

1.1 Der BTnode

Die BTnodes bestehen aus einem Atmel AVR-Mikrokontroller, einem Modul für Bluetooth und einem Modul für Low-power-Funkmodul, wie in der Abbildung 1.1 gezeigt wird. Sie bilden eine sehr kompakte, programmierbare Plattform, einsetzbar als mobiler Sensorknoten. Diese Plattformen eignen sich hervorragend, um Applikationen und neue Algorithmen im Bereich der Ad-Hoc-Netze zu implementieren. Der BTnode[1] wird an der ETH Zürich entwickelt: Am Institut für Theoretische Informatik (TIK) des Departements für Elektrotechnik und in der Gruppe Distributed System Group (DSG) des Departements für Informatik.

Nachfolgend werden die Hardware-Spezifikationen des BTnode aufgelistet:

Mikrokontroller Atmel ATmega 128L (7.3728 MHz @ 8 MIPS)

Memory 4 KB RAM, 4 KB EEPROM, 128 KB Flash-ROM, 256 KB externer SRAM

[1] http://www.btnode.ethz.ch

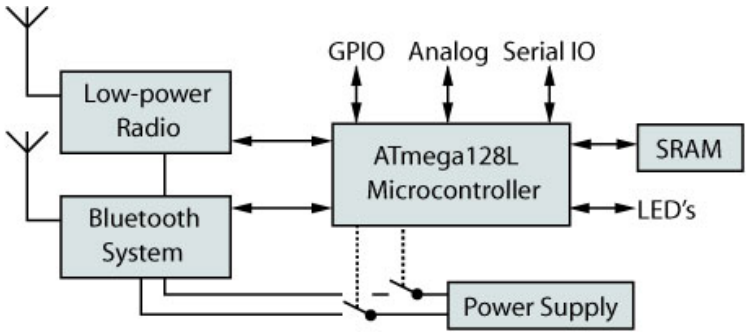


Abbildung 1.1: Das Komponentenschema des BTnode

Bluetooth Zeevo ZV4002, unterstützt Scatternetze mit max. 4 Piconetze/7 Slaves, kompatibel zur Bluetooth-Spezifikation 1.2

Low-power-Funkmodul Chipcon CC1000, Sendebereich im ISM Frequenzband 433-915 MHz

Externe Schnittstellen ISP, UART, SPI, I2C, GPIO, ADC, Timer, 4 LEDs

Das Bluetooth-Modul stellt im Vergleich zum Funkmodul eine hohe Bandbreite zur Verfügung, während das Funkmodul beim Betrieb weniger Strom verbraucht. Beide Module können gleichzeitig betrieben werden. Um den Stromverbrauch im Betrieb zu reduzieren, können beide Module unabhängig voneinander von der Spannungsquelle getrennt werden.

Als Betriebssystem für die BTnodes läuft ein Realtime-Embedded-OS mit dem Namen Nut/OS[2]. Basierend auf Ethernut ist es als Open-Source im Internet frei verfügbar. Dieses Betriebssystem bietet unter anderem folgende Eigenschaften:

- Non-preemtive Multithreading
- Events
- Einmalige und periodische Timer
- Dynamische Allokation von Heap-Speicher
- Interrupt-gesteuerte I/O

BTnut[3] ist die System-Software für die BTnodes, bestehend aus dem Betriebssystem Nut/OS, BTnode-spezifischen Treibern und einer Implementierung des Bluetooth-Stacks. Desweiteren beinhaltet BTnut eine Serie von Bibliotheken und eine ausführliche API-Dokumentation.

[2]http://www.ethernut.de/en/software.html
[3]http://www.btnode.ethz.ch/support/btnut_api/index.html

Um den BTnode mit neuem Programmcode zu betreiben, wird der neue Code im Flash-ROM gespeichert. Dieser Prozess wird Brennen genannt. Normalerweise wird dafür eine spezielle Hardware, der SPI-Programmer, verwendet.
Der Bootloader wird beim Aufstarten des BTnode als Erstes ausgeführt. Falls der Bootloader seine Aufgaben beendet hat, wird der eigentliche Programmcode gestartet.
Der Bootloader unterstützt zwei Methoden, um neuen Programmcode auf den BTnode zu brennen, ohne auf den SPI-Programmer zurückzugreifen. Erstens kann der Code beim Aufstarten über den seriellen Port on-the-fly auf den Flash-ROM verschoben werden. Zweitens wird beim Aufstarten im SRAM nach gültigem Programmcode gesucht. Falls der Code gefunden wurde, wird der Inhalt im Flash-ROM kopiert. Mehr Einzelheiten zu diesem Verfahren werden im Kapitel 4 auf Seite 23 erläutert.

1.1.1 Der Bluetooth-Stack im Überblick

Der Bluetooth-Stack ist eine standardisierte Implementierung eines Protokoll-Stacks. Zur Beschreibung der Struktur und Funktion von Protokollen für die Datenkommunikation über Bluetooth liegt das Architekturmodell in Abbildung 1.2 zugrunde. Der Stack besteht aus mehreren Schichten und jede dieser Schichten definiert gewisse Funktionen, die beim Austausch von Daten zwischen Anwendungen über die darunterliegenden Schichten ausgeführt werden. Die Schnittstellen zwischen den Schichten sowie die Protokolle sind standardisiert und offen (d.h. herstellerunabhängig und frei zugänglich). Der Stack bietet damit auch eine einheitliche Schnittstelle zwischen Hardware und Software an.

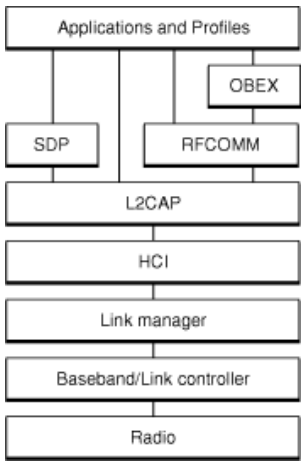


Abbildung 1.2: Der Bluetooth-Stack

Das Host Controller Interface (HCI) bietet eine Befehlsschnittstelle zum

Basisbandkontroller und Linkmanager. Der HCI-Layer des Stacks kommuniziert mit der Bluetooth-Hardware und bildet eine einheitliche Schnittstelle zwischen Software und Hardware. Eine der wichtigsten Aufgaben ist das Auffinden von Bluetooth-Geräten, die sich in Reichweite befinden. Diese Funktion wird als Inquiry oder als Device Discovery bezeichnet.

Das Logical Link Control and Adaptation Protocol (L2CAP) bietet höherwertigen Protokollen verbindungsorientierte und verbindungslose Datendienste an. Dazu gehören Protokollmultiplexing, Segmentierung und Reassemblierung. L2CAP erlaubt höherwertigen Protokollen und Programmen den Versand und Empfang von L2CAP-Datenpaketen mit einer Länge von bis zu 64 Kilobytes. L2CAP arbeitet kanalbasiert. Ein Kanal ist eine logische Verbindung innerhalb einer Basisbandverbindung. Mehrere Kanäle können sich deshalb die gleiche Basisbandverbindung teilen. Die Kanäle werden mit einem PSM (Protocol Service Multiplexer) adressiert.

Das Radio Frequency Communication Protocol (RFCOMM) emuliert serielle Verbindungen über das L2CAP-Protokoll. Bei RFCOMM handelt es sich um ein einfaches, streambasiertes Transportprotokoll. RFCOMM unterstützt Anwendungen, die auf serielle Ports angewiesen sind.

Das Service Discovery Protocol (SDP) erlaubt es Clientanwendungen, von Serveranwendungen angebotene Dienste sowie deren Eigenschaften abzufragen. Zu diesen Eigenschaften gehören die Art oder die Klasse der angebotenen Dienste sowie der Mechanismus oder das Protokoll, die zur Nutzung des Dienstes notwendig sind. SDP bietet aber keine Mechanismen zur Verwendung dieser Dienste. Normalerweise sucht ein SDP-Client nach Diensten, die bestimmte geforderte Eigenschaften erfüllen. Diese Funktion wird als Service-Discovery bezeichnet. Es ist aber auch möglich, eine allgemeine Suche durchzuführen, welche alle verfügbaren Dienste auflistet. Diese Funktion wird Browsing genannt.

Das Object Exchange Protocol (OBEX) ist ein häufig verwendetes Protokoll für den Dateitransfer zwischen Mobilgeräten. Sein ursprünglicher Anwendungsbereich war die Kommunikation über die Infrarotschnittstelle. Jetzt ist es allgemein auf drahtlose Schnittstellen wie u.a. Bluetooth anwendbar.

Das Bluetooth-Modul auf dem BTnode, der Zeevo ZV4002, stellt eine Firmware bereit, welche den HCI-Layer implementiert. Momentan sind die HCI-, L2CAP- und RFCOMM-Protokolle in BTnut implementiert. Eine Unterstützung des SDP-Dienstes ist für die Zukunft geplant[4].

[4] http://sourceforge.net/tracker/index.php?
func=detail&aid=1234289&group_id=81773&atid=563980

1.1.2 JAWS als Ad-Hoc-Netz-Anwendung

Der Schwerpunkt einer Sensornetz-Anwendung liegt in der Formation und Wartung des Ad-Hoc-Netzes, das eine möglichst fehlerlose und stabile Datenübertragung zur Verfügung stellt. Um eine solches Netzwerk mit Bluetooth zu realisieren, werden Formationen über mehrere Piconetze[5], ein sogenanntes Scatternetz, aufgebaut. Abbildung 1.3 illustriert den Aufbau eines Scatternetzes. JAWS[6] beinhaltet einen Algorithmus, um automatisch ein Baumnetzwerk für die Multihop-Kommunikation[7] aufzubauen.

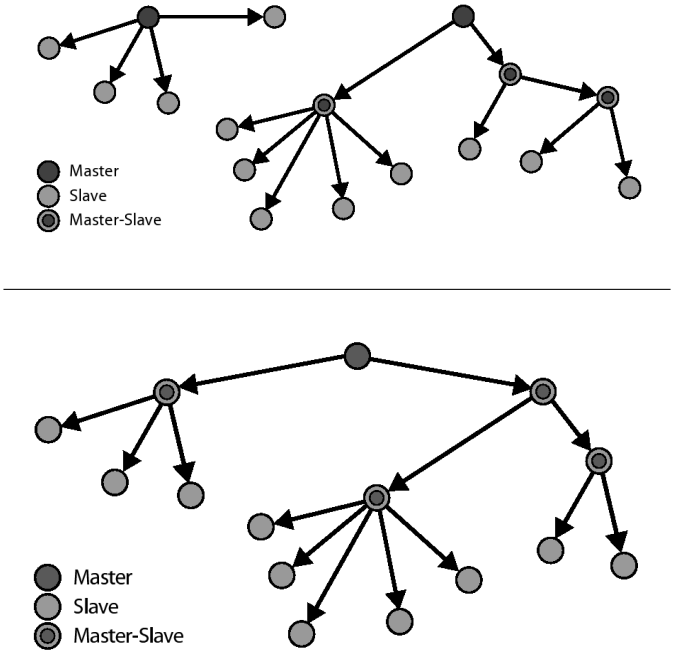


Abbildung 1.3: Die Aufbauphase eines Bluetooth-Scatternetzes

1.2 Der PDA: HP iPAQ h5450

Die Arbeit wurde auf einem iPAQ h5450 ausgeführt. Dieser PDA bietet eine akzeptable Rechenleistung, ein Farbdisplay und 64 MB RAM. Notwendig ist ein Onboard-Bluetooth für die Kommunikation mit dem BTnode. Noch wichtiger ist die Tatsache, dass das Modell von der Linux-Distribution Familiar inklusive der Bluetooth-Hardware unterstützt wird. Informationen über die Unterstützung aller PDAs und TabletPCs von Familiar findet man im

[5]http://de.wikipedia.org/wiki/Bluetooth, Systemarchitektur
[6]http://www.btnode.ethz.ch/projects/jaws
[7]http://de.wikipedia.org/wiki/Unicast

handheld.org-Wiki[8]. Desweiteren wird der PDA mit einer SD-Karte (128 MB) bestückt, um genügend Speicherkapazität für Programmpakete zur Verfügung zu stellen.

Abbildung 1.4: Die Front-Ansicht des HP iPAQ h5450

Nachfolgend die Hardware-Spezifikationen des iPAQ h5450:

- 400MHz PXA250 XScale Prozessor
- Transreflektives Farbdisplay (240 x 320 Pixel, 65536 Farben)
- 64MB RAM / 48MB ROM
- Bluetooth & 802.11b Wi-Fi
- Austauschbarer Li-Polymer Akku
- Long Range IrDA Port
- Secure Digital Slot mit SDIO Support

Auf dem PDA wird eine Linux-Distribution namens Familiar betrieben. Linux bietet mit dem Bluetooth-Stack BlueZ eine gute Unterstützung für Bluetooth-Dienste an. Für Java wird das standardisierte Bluetooth-API angeboten. Allfällige Portierungsarbeiten für den PDA werden mit der Crosscompiler-Toolchain OpenEmbedded ausgeführt.

[8]http://www.handhelds.org/moin/moin.cgi/SupportedHandheldSummary

1.2.1 Linux für den PDA: Familiar

Die Familiar-Distribution[9] basiert auf der bekannten Debian-Distribution[10]. Analog zu Debian ist bei Familiar ein Package-Management-System vorhanden. Itsy Package (IPKG) lädt von einem Internet-Repository mit

```
ipkg update
```

eine Liste mit der Beschreibung der aktuellen Pakete herunter und bringt das System mit

```
ipkg upgrade
```

automatisch auf den neuesten Stand. Neue Pakete wie z.B. der Texteditor `nano` werden mit

```
ipkg install nano
```

vom Internet heruntergeladen und installiert, bei Nichtgebrauch mit

```
ipkg remove nano
```

entfernt. Abhängigkeiten zu anderen Paketen werden erkannt und bei Bedarf werden zusätzlich notwendige Pakete mitinstalliert. IPKG-Pakete tragen die Endung `ipk`.
Enthalten ist eine grafisch-ansprechende Oberfläche mit typischen PDA-Anwendungen, wie z.B. ein Terminkalender. Familiar unterstützt zwei grafische Fenstersysteme: GPE und Opie. GPE[11] benutzt analog zu Gnome das freie Toolkit GTK+. GTK+[12] ist eine Komponentenbibliothek unter der LGPL-Lizenz, mit welcher grafische Benutzeroberflächen (GUI) für Softwareprogramme erstellt werden können. Opie[13] benutzt analog zu KDE das Toolkit Qt. Qt[14] ist eine Klassenbibliothek und Entwicklungsumgebung unter GPL-Lizenz oder kommerzieller Lizenz für die plattformübergreifende Programmierung grafischer Benutzeroberflächen. Es wird von der norwegischen Firma Trolltech entwickelt und ist für verschiedene Betriebssysteme wie Linux, Mac OS X und Windows erhältlich. Leider sind beide Toolkits nicht zueinander kompatibel, so dass eine grafische Applikation normalerweise entweder auf GPE oder auf Opie läuft.
Zukünftig ist aber eine weitere Einbindung des Toolkits von Enlightenment[15] geplant, welches einen Wrapper für GTK+ und Qt besitzen soll. Damit entfällt die Entscheidung des Applikationsentwicklers für ein Toolkit. Dank der flexiblen Konfigurationsstruktur ist Enlightenment nicht nur auf Funktionen zur Verwaltung von Fenstern beschränkt: Es ist u.a. möglich, von der typischen Desktop-Anwendung abzuweichen und neue Wege für die Navigation zu implementieren (wie in Abbildung 1.5 rechts auf Seite 8 illustriert wird).

[9] http://familiar.handhelds.org
[10] http://www.debian.org
[11] http://gpe.handhelds.org
[12] http://de.wikipedia.org/wiki/GIMP-Toolkit
[13] http://opie.handhelds.org
[14] http://de.wikipedia.org/wiki/Qt-Toolkit
[15] http://www.enlightenment.org/

Abbildung 1.5: Bildschirmfotos von GPE, Opie und Enlightenment (DR17)

Nachfolgend eine kleine Zusammenstellung der Eigenschaften von Familiar:

- Wahl zwischen zwei grafischen Umgebungen, beide mit voll ausgestatteter Personal-Information-Management-Softwareumgebung
  - GPE
  - Opie
- Unterstützung eines Package-Management-Systems
- Viele Systemprogramme sind in einer einzigen Applikation implementiert, was zu Einsparung von Speicherplatz führt.
- Standardmässig ist ein SSH-Server dabei.
- Unterstützung der gebräuchlichen Netzwerk-Protokolle, wie SSH, PPP, IPsec, MIPv6 etc.
- Die ganze Distribution wird mit einer einzigen Toolchain kompiliert und aufgebaut.

Erwähnenswertes

- Eine englische Anleitung für die Installation von Familiar auf den PDA findet man im Internet[16]. Beim Flashen hat man die Wahl zwischen GPE oder Opie: In dieser Arbeit fiel die Entscheidung auf GPE.

- Um den PDA mit einer Workstation zu verbinden, hat man die Wahl zwischen der seriellen Schnittstelle, USB, WLAN oder Bluetooth. Um das Personal Area

[16] http://familiar.handhelds.org/releases/v0.8.2/install

Network[17] (PAN) über Bluetooth zu nutzen, legt man auf der Workstation die Datei `/etc/bluetooth/pan/dev-up` mit folgendem Inhalt an:

```
#!/bin/sh
# 'dev-up' script on NAP
ifup $1
```

Nun ist die statische Netzwerkkonfiguration in `/etc/network/interfaces` für die Schnittstelle `bnep0` zu konfigurieren. Die Verbindung kann auf dem iPAQ mit

```
# xx:xx:xx:xx:xx:xx durch die Hardware Adresse vom
# Bluetooth Device der Workstation (Bluetooth) ersetzen.
pand --role PANU --connect xx:xx:xx:xx:xx:xx
```

aufgebaut werden. Eine detailliertere Anleitung[18] zu PAN befindet sich auf dem Webserver von BlueZ.

- Es besteht die Möglichkeit, ein Backup zu generieren, welches direkt auf einem iPAQ geflasht werden kann. Eine ältere Anleitung findet man im Wiki[19] von Familiar. Aus Gründen der Aktualität befindet sich im Anhang auf Seite 45 das - auf den iPAQ h5450 angepasste - Skript `backup.sh`.

1.2.2 Die Java-VM für Familiar

Familiar unterhält in seinem Paket-Repository drei Java-Virtual-Machines: SableVM, JamVM und Kaffe. Kaffe[20] ist im aktuellen Familiar nicht vollständig portiert bzw. lauffähig und wird daher nicht weiter behandelt. SableVM[21] und JamVM[22] basieren auf dem GNU-Classpath[23] Projekt, einer freien Implementierung der Hauptklassen von Sun (hauptsächlich die `java.*`- und `javax.*`-Klassen). Auf der Kommandozeilen-Ebene sollten sie sich also beide gleich verhalten. Unterschiede zeigen sich im AWT- und Swing-Support. Swing läuft auf dem iPAQ sehr langsam und instabil, worauf wir uns auf AWT beschränkt haben. SableVM scheint aber punkto AWT besser und stabiler zu laufen.
Die in den nächsten Kapiteln beschriebenen Java-Programme laufen auf dem iPAQ auf *SableVM*. Es wird empfohlen, auf der Workstation die gleiche JVM wie auf dem PDA zu benutzen. Bugs und fehlende Komponenten können so früher entdeckt werden.

[17] http://de.wikipedia.org/wiki/Personal_Area_Network
[18] http://bluez.sourceforge.net/contrib/HOWTO-PAN
[19] http://handhelds.org/moin/moin.cgi/FamiliarBackupHowto
[20] http://www.kaffe.org
[21] http://sablevm.org
[22] http://jamvm.sourceforge.net
[23] http://www.gnu.org/software/classpath/home.html

1.2.3 Das Bluetooth-API für Java: JSR-82

JSR-82[24] stellt das Standard-API für die Entwicklung von Bluetooth-Anwendungen und -Profilen mit der Programmiersprache Java bereit und ermöglicht einen standardisierten Zugriff auf die Funktionalitäten von Bluetooth. Das API wurde ursprünglich für Bluetooth-Mobiltelefone mit J2ME[25] entworfen. JSR-82 stellt selber keine Implementierung zur Verfügung. Es existieren aber vorinstallierte Implementationen für J2ME auf Mobiltelefonen und einige Implementationen für J2SE auf unterschiedlichen Betriebssystemen. Das Buch "Bluetooth Application Programming with the Java APIs" empfiehlt sich für das Studium in das JSR-82-API.
JSR-82 unterstützt Device Discovery sowie Service Discovery und bietet unter anderem eine Schnittstelle für die Datenkommunikation bzw. den Dateitransfer über L2CAP (nur verbindungsorientiert), RFCOMM und OBEX an.

Die Suche nach einer JSR-82 Implementierung für Linux

Zur Zeit existieren zwei Implementierungen für Linux: eine von Rococo[26] und eine von Avetana[27]. Das Impronto Developer Kit von Rococo ist nur für Linux und PalmOS verfügbar und wurde in letzter Zeit nicht weiterentwickelt. Besser sieht es bei der Implementierung von Avetana aus: AvetanaBT ist für Windows, MacOS X und Linux verfügbar, die Linux-Implementierung[28] steht als Open-Source unter der GPL-Lizenz und kann von Sourceforge heruntergeladen werden. AvetanaBT greift auf BlueZ zurück. Das offizielle Bluetooth-Subsystem BlueZ[29] von Linux stellt seit vier Jahren eine stabile und offene Implementierung des Bluetooth-Stack zur Verfügung. Wir werden uns in dieser Arbeit auf *AvetanaBT* konzentrieren.

1.2.4 Die Toolchain: OpenEmbedded

Ein PDA eignet sich mit seinen knapp bemessenen Ressourcen nicht oder nur sehr bedingt zum Kompilieren und Linken komplexer Programme. Die Programme werden auf einer leistungsfähigeren Maschine wie z.B. auf dem PC kompiliert. Hierzu bedarf es einer so genannten Crosscompiler[30]-Toolchain, die den plattformabhängigen, fremden Maschinencode generiert. Bis zum fertigen `ipk`-Paket werden aber zuvor alle möglichen Bibliotheken erzeugt. Die sich dar-

[24] http://www.jcp.org/en/jsr/detail?id=82
[25] http://de.wikipedia.org/wiki/J2ME
[26] http://www.rococosoft.com/DK_release_notes.html
[27] http://www.avetana-gmbh.de/avetana-gmbh/produkte/jsr82.xml
[28] http://sourceforge.net/projects/avetanabt
[29] http://www.bluez.org
[30] http://de.wikipedia.org/wiki/Crosscompiler

aus ergebenden Abhängigkeiten und verschiedene Umgebungsvariablen müssen angepasst werden. OpenEmbedded[31] fasst alle Schritte zu einer einheitlichen Lösung zusammen. OpenEmbedded unterhält u.a. ein Archiv von Metadaten für Bitbake. Bitbake[32] leistet vom Quellcode an bis zum fertigen Paket die eigentliche Arbeit. Die Metadaten von OpenEmbedded beinhalten Direktiven, mit deren Hilfe Bitbake einzelne Programme fertigt. Sie berücksichtigen die dazu passenden Quellcode-URLs sowie die dazu anzubringenden Patches, die Abhängigkeiten zu anderen Programmen und spezielle Installationsroutinen. Desweiteren ist OpenEmbedded in der Lage, eigene Distributionen für fast beliebige Zielplattformen zu schnüren. Es eignet sich übrigens nicht nur für PDAs, sondern ebenso für andere Embedded-Geräte. OpenEmbedded hat sich in der Praxis als derart leistungsfähig erwiesen, dass mittlerweile auch die Entwickler anderer Embedded-Plattformen wie OpenZaurus[33], Familiar und OpenSIMpad[34] ihre Distributionen mit dieser Toolchain erstellen.

Auf die Installation und Konfiguration von OpenEmbedded und Bitbake werde ich nicht weiter eingehen, eine gute Wiki-Anleitung[35] ist vorhanden. Leider ist die Dokumentation im Wiki zur Struktur der Metadaten recht löchrig. Am besten findet man beim Studium der vorhandenen Programmpakete Ansatzpunkte für die Konstruktion eines neuen Pakets.

Der Prozess bis zum fertigen Paket besteht aus sechs Phasen:

fetch Die Quelltexte werden vom Internet heruntergeladen. Als Übertragungsprotokolle werden für Archive HTTP oder FTP, für Quelltexte unter Versionskontrolle CVS oder SVN unterstützt.

patch Die Quelltexte können für die Weiterverarbeitung gepatcht werden.

configure Die Quelltexte werden - falls nötig - konfiguriert.

compile Die Quelltexte werden kompiliert.

staging Dateien, welche von anderen Paketen benötigt werden (Bibliotheken, Header-Files etc.), werden in ein eigenes Verzeichnis installiert.

package Das Paket für IPKG wird erstellt.

Im Header der Metadatei stehen allgemeine Informationen, Abhängigkeiten zu anderen Paketen, die URL zu den Sourcen etc. Über `inherit` kann man auf Standard-Routinen zurückgreifen. Ein Beispiel: Falls "GNU autotools" [4] im Projekt verwendet wird, kann `inherit autotools` angegeben werden. Die

[31] http://oe.handhelds.org
[32] http://bitbake.berlios.de/manual
[33] http://www.openzaurus.org
[34] http://www.opensimpad.org
[35] http://oe.handhelds.org/cgi-bin/moin.cgi/GettingStarted

Phasen **configure**, **compile**, **staging** und **package** werden so automatisiert. Leider ist die Dokumentation zu diesen Routinen unvollständig. Als Ersatz können die Dateien im `classes`-Verzeichnis näher angeschaut werden.

Das Bereitstellen einiger Software-Pakete

Subversion[36] ist analog zu CVS eine Software zur Versionsverwaltung. Subversion greift auf "GNU autotools" zurück, die Paketierung war damit nicht besonders aufwendig.
AvetanaBT wurde als nächstes für den PDA kompiliert. Normalerweise wird zum Kompilieren das Java SDK von Sun benutzt. Für OpenEmbedded wurden stattdessen freie, bereits im OpenEmbedded vorhandene Komponenten benutzt (d.h. `javac`, `jar` und `javah` wurden durch freie Programme wie `jikes`, `fastjar` und `kaffeh` ersetzt). In einigen Java-Methoden wird für die Implementierung auf BlueZ-Bibliotheken und damit auf die C-Programmiersprache zurückgegriffen. Für die Verfügbarkeit des Java-Native-Interfaces (JNI) wurden die Pakete `kaffeh` und `classpath` in den Kompilationsprozess eingebunden.
Für die Demo in der Präsentation und für Bilder im Bericht wurde zudem mit *fbVNCServer* ein VNC[37]-Server für den iPAQ bereitgestellt. Damit kann der Bildschirminhalt des PDA mit einem VNC Client auf der Workstation angezeigt und über Maus und Tastatur ferngesteuert werden. Die Bildschirmfotos werden einfacherweise mit `vncsnapshot`[38] angefertigt.
Alle drei Metadateien sind im Anhang auf Seite 45 zu finden. Sie wurden bereits offiziell ins OpenEmbedded-Repository aufgenommen.

[36] http://de.wikipedia.org/wiki/Subversion_(Software)
[37] http://de.wikipedia.org/wiki/Vnc
[38] http://vncsnapshot.sourceforge.net

Kapitel 2

Der BTnodeInspector

Die Software für den PDA ist in der Programmiersprache Java geschrieben und nutzt AWT als grafische Schnittstelle. In dieser Arbeit entstanden drei GUI-Applikationen:

RemoteProgramming Der BTnode kann über Bluetooth bzw. über L2CAP neu programmiert werden.

AttributeRetrieving Attribute auf dem BTnode werden über Bluetooth abgefragt.

TopologyDrawing Die Topologie eines JAWS-Netzes wird grafisch dargestellt.

Der BTnodeInspector fungiert als Rahmenprogramm für die drei Kernapplikationen und läuft auf einer beliebigen, J2SE-fähigen Plattform. Für die Bluetooth-Kommunikation mit dem BTnode wird ausschliesslich das paketorientierte L2CAP verwendet.

Eine kurze Beschreibung der Komponenten der BTnodeInspectors aus dem Blockschema 2.1:

JSR-82 JSR-82 ist das Standard-API für die Bluetooth-Kommunikation.

L2CAPCommunication `L2CAPCommunication` ist ein Framework für die Kommunikation über L2CAP. `L2CAPCommunication` bietet eine Timeout-Implementierung, eignet sich damit auch für asynchrone Kommunikationsabläufe. Mehr dazu im Kapitel 3 auf Seite 17.

LocalBluetoothDevice `LocalBluetoothDevice` ist hauptsächlich für das Auffinden von weiteren Bluetooth-Geräten zuständig. Mehr dazu im Unterkapitel 3.2 auf Seite 18.

RemoteProgrammer `RemotePogrammer` ist ein Modul zur Neuprogrammierung der BTnodes über L2CAP. `RemoteProgrammer` bietet unter anderem eine Schnittstelle für die Kommandozeile an. Mehr dazu im Kapitel 4 auf Seite 23.

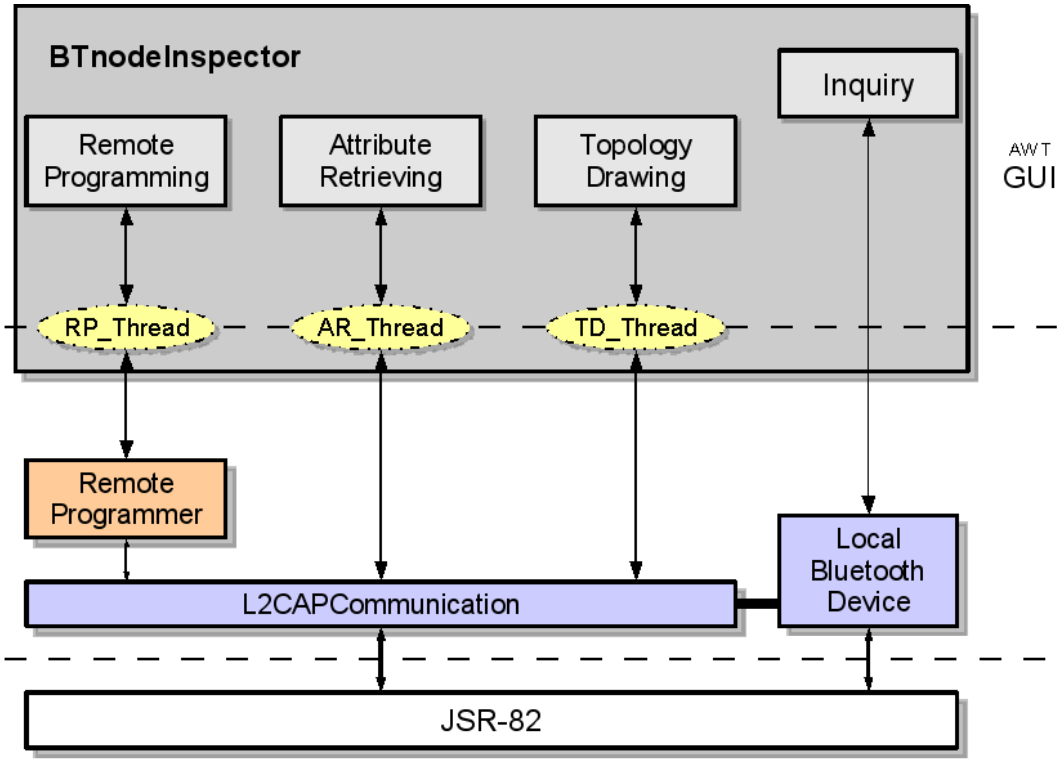


Abbildung 2.1: Das Blockschema des BTnodeInspector

Threads Sie fungieren als Schnittstelle zwischen Logikschicht (bzw. Kommunikationsschicht) und grafischer Oberfläche und haben zwei Hauptaufgaben: Reagieren auf Aktionen im GUI und Aktualisierung des GUI mit neuen Daten aus der Kommunikationsschicht.

Inquiry Eine Fenstereinheit im BTnodeInspector zum Suchen, Filtern und Selektieren von BTnodes. Sie greift auf die Klasse `LocalBluetoothDevice` zu.

RemoteProgramming `RemoteProgramming` ist eine grafische Erweiterung des Moduls `RemoteProgrammer`. Mehr dazu im Kapitel 4.4 auf Seite 25.

AttributeRetrieving `AttributeRetrieving` listet verfügbare Informationen von einem BTnode auf. Gewünschte bzw. selektierte Informationen werden angezeigt und laufend aktualisiert. Mehr dazu im Kapitel 5 auf Seite 29.

TopologyDrawing `TopologyDrawing` bietet eine grafische Darstellung der Baumtopologie eines JAWS-Netzwerkes. Mehr dazu im Kapitel 6 auf Seite 35.

Der BTnodeInspector macht generell nichts anderes, als ein Inquiry auszuführen, BTnode-Kandidaten vom Benutzer selektieren zu lassen und im Nachhinein den Konstruktor der gewählten Kernapplikation mit den selektierten BTnodes als Parameter aufzurufen. Kandidaten sind je nach Anwendung BTnodes

allgemein (für RemoteProgramming oder AttributeRetrieving) oder JAWS-Knoten (für TopologyDrawing). Die Selektion funktioniert entweder einzeln (für AttributeRettrieving oder TopologyDrawing) oder mehrfach (für Remote-Programming). Mit dem 'Start'-Button wird die Applikation aufgebaut. Unter 'Help' in der Menüleiste können zudem Informationen über das Bluetooth-Gerät eingesehen werden. Abbildung 2.2 auf Seite 15 zeigt einige Bildschirmfotos der Oberfläche.
Der modulare Aufbau der Kernapplikationen mit dem Aufruf über das Rahmenprogramm hat den Vorteil, dass die Kernapplikationen auch einzeln in ein anderes Programm eingebettet werden können.

Abbildung 2.2: Bildschirmfotos vom BTnodeInspector: links mitten in der Inquiryphase, rechts wird AttributeRetrieving angewählt

Kapitel 3

L2CAPCommunication

Für die L2CAP-Kommunikation wurde ein eigenes Framework entwickelt. Dies basiert auf einigen Argumenten:
Die Bedienung und Anwendung soll vereinfacht werden. Das JSR-82-API ist teilweise schwierig gestaltet und auf den ersten Blick nicht leicht verständlich. Als Beispiel die Methode zum Verbinden: Angaben wie Ziel oder Optionen werden in einem einzigen String spezifiziert. Eine Methode, welche diese Angaben direkt als Parameter fordert, würde die Bedienung vereinfachen.
Desweiteren gibt es Methoden, welche den Programmablauf blockieren können. Es ist sinnvoll, insbesondere für eine nicht-deterministische Kommunikation, mindestens ein Timeout zu berücksichtigen. Die Kommunikation kann natürlich auf unterschiedliche Weise implementiert werden. Daher wird mit erweiterten Instanzen der Basisklasse `javax.bluetooth.L2CAPConnection` gearbeitet. Im Kapitel 3.3 auf Seite 19 wird darauf näher eingegangen.
Alle Klassen bieten Debug-Dienste an, welche bei Bedarf eingeschaltet werden können. Sie schildern den Kommunikationsablauf, den Inhalt der Pakete und zeigen weitere, hilfreiche Informationen an.
Eine kleine Bemerkung am Rande: Dieses Framework ist auch auf J2ME lauffähig. Da aber Meldungen über `System.out.print` ausgegeben werden, ist der Debug-Dienst auf dem Mobiltelefon nicht verfügbar.

3.1 L2CAPCommunication

`L2CAPCommunication` gebraucht das Singleton-Pattern[1]. Dieses stellt sicher, dass zu einer Klasse nur genau ein Objekt erzeugt wird und ermöglicht einen globalen Zugriff auf dieses Objekt. Aus Sicht einer Applikation wird es folgendermassen eingebunden:

[1] http://de.wikipedia.org/wiki/Singleton

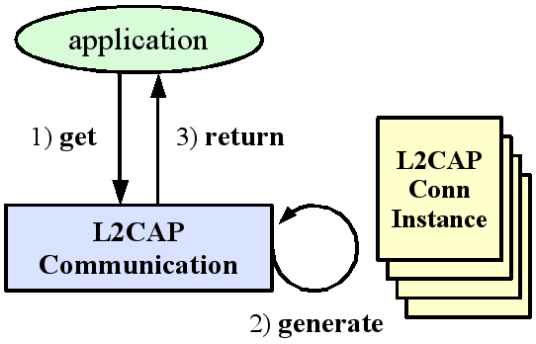


Abbildung 3.1: Die Interaktion mit L2CAPCommunication

```
// (initializes and) references L2CAPCommunication
L2CAPCommunication comm =
        L2CAPCommunication.getInstance();
```

Abbildung 3.1 zeigt den weiteren Ablauf. Zur Generierung einer Instanz der L2CAP-Verbindung wird das Factory-Design-Pattern angewendet. Die Applikation ruft dabei folgende Zeile auf:

```
// generates and returns instance of L2CAPConnection
L2CAPConnInstance conn = comm.getL2CAPConnInstance();
```

Ein kleines Beispiel für den Gebrauch der Instanz `conn`:

```
// PSM 0x1001 is the echo service port
conn.doConnect("00043F000092", "1001");
System.out.println(new String(
        conn.doSendRecv("Hello World!".getBytes())));
conn.doDisconnect();
```

3.2 LocalBluetoothDevice

`LocalBluetoothDevice` stellt u.a. Informationen über das Bluetooth-Gerät zur Verfügung. `LocalBluetoothDevice` gebraucht das Singleton-Pattern und ist eine Erweiterung der JSR-82-Klasse `javax.bluetooth.LocalDevice`. Mehrere Objekte, die das `javax.bluetooth.DiscoveryListener`-Interface implementieren, können durch das Observer-Pattern am `LocalBluetoothDevice` horchen. Momentan ist die Klasse nur auf Device Discovery bzw. auf Inquiry ausgelegt, da der Service Discovery in dieser Arbeit nicht verwendet wurde.

3.3 L2CAPConnInstance

Die Methode zum Verbinden über L2CAP wird für das Framework vereinfacht. Im JSR-82-API wird die Addressierung über eine URL angegeben:

```
L2CAPConnection conn = (L2CAPConnection) Connector.open(
        "btl2cap://00043F000092:1001" +
            ";ReceiveMTU=672;TransmitMTU=672");
```

Im URL wird als Erstes das Kommunikationsprotokoll mit `btl2cap` spezifiziert. Die Hardware-Adresse und der PSM werden vor bzw. nach dem Doppelpunkt übergeben. Nach der PSM werden optionale Angaben, wie z.B. die maximale Übertragungseinheit (MTU), angehängt. Die Optionen werden jeweils durch Semikolons getrennt. Einfacher wäre hier eine Connect-Methode, die die nötigen Angaben als Funktionsparameter fordert.
JSR-82 besitzt desweiteren für L2CAP drei primitive Kommunikationselemente:

- `void L2CAPConnection.send(byte[] b)`
- `boolean L2CAPConnection.ready()`
- `int L2CAPConnection.receive(byte[] b)`

`receive` gibt die Länge des empfangenen Pakets zurück. Da die Länge in Java über `b.length` abgefragt werden kann, ist es intuitiver, direkt den Inhalt des Paketes als Bytearray mit passender Längenallokation zurückzugeben. Desweiteren soll ein Timeout implementiert werden, da der Aufruf von `receive` den Programmablauf blockiert.

Die obengenannten Punkte werden im Framework berücksichtigt. Damit die Instanzen der L2CAP-Verbindung möglichst flexibel erweiterbar und trotzdem einfach bedienbar sind, wird die Architektur in Abbildung 3.2 auf Seite 20 angewendet.

Das Interface `L2CAPConnInstance` definiert einen Minimalsatz an Methoden:

```
void doConnect(String mac, int psm)
void doConnect(String mac, String psm)
void doConnect(String mac, int psm, int mtu)
void doConnect(String mac, String psm, int mtu)
void doSend(byte[] b)
byte[] doRecv()
byte[] doRecv(int timeout)
byte[] doSendRecv(byte[] b)
byte[] doSendRecv(byte[] b, int timeout)
void doDisconnect()
void setDebugMode(boolean debugMode)
boolean getDebugMode()
```

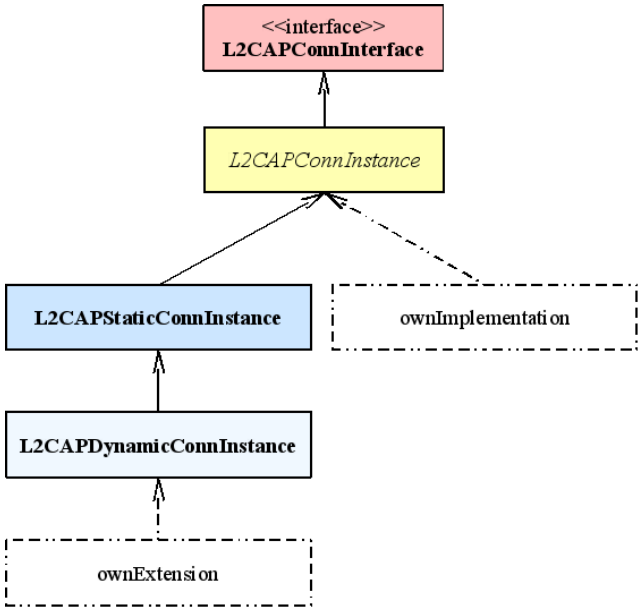


Abbildung 3.2: Die L2CAPConnInstance-Architektur

Damit wird einerseits das Ganze übersichtlich, anderseits wird eine gewisse Kompatibilität zu anderen Implementierungen gewährleistet. Die abstrakte Klasse `L2CAPConnInstance` implementiert `L2CAPConnInterface`. Die Klasse gilt als Basisklasse für alle Implementierungen einer L2CAP-Kommunikation. In dieser Klasse werden zudem Standardwerte für die MTU und den Timeout definiert.

Zwei Implementierungen sind bereits verfügbar: `L2CAPStaticConnInstance` und `L2CAPDynamicConnInstance`. Beide Implementierungen besitzen die gleiche Methode, um das Timeout zu implementieren: Es wird mit der JSR-82-Methode `ready` gepollt, mit der man feststellen kann, ob Daten im Empfangsbuffer liegen. Falls Daten verfügbar sind, kann ohne Gefahr einer Blockierung `receive` benutzt werden.

Zu den Unterschieden der beiden Implementierungen: Abbildung 3.3 zeigt zwei verschiedene Anwendungsbereiche. `L2CAPStaticConnInstance` ist für eine Kommunikation geeignet, deren Verbindung während des Gebrauchs nicht gewechselt wird. Beim häufigen Wechsel der Verbindung ist `L2CAPDynamicConnInstance` zu nehmen. In der Implementierung wird verhindert, dass bei jedem Wechsel eine zusätzliche Verbindung aufgebaut wird. Die alte Verbindung wird bei Notwendigkeit automatisch abgebaut.

`L2CAPStaticConnInstance` ist für die Verbindung zwischen dem Client (z.B. Workstation oder PDA) und einem Endgerät (z.B. BTnode) gedacht. Es implementiert im Prinzip das, was im Interface `L2CAPConnInterface` beschrieben ist. Die Implementierung wird über den Identifikator `STATIC_CONNECTION` spezifiziert:

Abbildung 3.3: Anschauung der L2CAPConnInstance-Implementierungen: links L2CAPStaticConnInstance, rechts L2CAPDynamicConnInstance

```
// create instance of L2CAPConnection (static)
L2CAPStaticConnInstance sconn =
        (L2CAPStaticConnInstance) comm.getL2CAPConnInstance(
                L2CAPCommunication.STATIC_CONNECTION);
```

`L2CAPDynamicConnInstance` ist für eine Verbindung zwischen dem Client mit mehreren, potentiellen Endgeräten gedacht. Der Aufruf der Methoden `doConnect` und `doDisconnect` beim Wechsel der Verbindung ist nicht mehr nötig. Diese Implementierung vereinfacht damit den Verbindungsaufbau und -abbau und ergänzt sich mit zusätzlichen Methoden:

```
void doSend(String mac, String psm, int mtu, byte[] b)
byte[] doRecv(String mac, String psm, int mtu)
byte[] doRecv(String mac, String psm, int mtu, int timeout)
byte[] doSendRecv(String mac, String psm, int mtu, byte[] b)
byte[] doSendRecv(String mac, String psm, int mtu, byte[] b, int timeout)
```

`L2CAPDynamicConnInstance` ist eine Erweiterung der Klasse `L2CAPStaticConnInstance`. Falls `doSend` oder `doRecv` ohne Zielangaben aufgerufen werden, werden diejenigen der letzten Verbindung übernommen. Die Implementierung wird über den Identifikator `DYNAMIC_CONNECTION` spezifiziert:

```
// create instance of L2CAPConnection (dynamic)
L2CAPDynamicConnInstance dconn =
        (L2CAPDynamicConnInstance) comm.getL2CAPConnInstance(
                L2CAPCommunication.DYNAMIC_CONNECTION);
```

Es ist zudem möglich, z.B. auf die Basisklasse `L2CAPConnInstance` eigene Implementierungen hinzuzufügen oder `L2CAPDynamicConnInstance` mit einer weiteren Klasse zu erweitern, wie in Abbildung 3.3 illustriert wird.

Kapitel 4

RemoteProgramming

Bis jetzt gab es drei Möglichkeiten, einen BTnode neu zu programmieren: über den SPI-Programmer, über den seriellen Port oder über den SRAM. In dieser Arbeit wurde die Funktionalität um eine weitere Methode erweitert: Der BTnode kann über eine Bluetooth-Verbindung neu programmiert werden. Der Programmcode wird an den BTnode übermittelt und der BTnode legt ihn als Abbild im SRAM ab. Sobald die Übermittlung beendet ist, startet der BTnode neu und der Bootloader erledigt den Rest: Er findet das Abbild im SRAM und beschreibt damit den Flash-ROM neu. Dieser neue Service wird als RemoteProgramming bezeichnet. Mit dem mitgeliefertem Java-Client kann ein BTnode in Bluetooth-Reichweite neu programmiert werden.

4.1 Der Bootloader der BTnodes

Der Bootloader unterstützt das Brennen über ein Abbild im SRAM. Damit der Bootloader ein gültiges Abbild als solches erkennt, wird folgender Header verwendet:

prog_type	prog_ver	aktiv_flag	prog_len	CRC	...data...	CRC
1	4	1	4	2	prog_len	2

In der unteren Zeile wird die Länge der einzelnen Felder in Bytes angegeben. `prog_type` und `aktiv_flag` müssen mit speziellen Bytes versehen werden und die beiden Checksummen `CRC` müssen stimmen (das erstere über den Header bzw. über die ersten zehn Bytes, das zweite über `prog_len`), damit der Flash-ROM vom Bootloader beschrieben wird. Die Flash-Daten `data` bestehen aus mindestens einem Block mit folgender Struktur:

block_length	block_start_address	...block_data...
3	3	block_length

Ein Block besteht aus einer Startadresse und Daten mit einer bestimmten Länge. Mehrere Blöcke werden gebraucht, falls es Lücken beim Beschreiben des Flash-ROM gibt und damit Adresssprünge auftauchen.

4.2 Der Kommunikationsablauf zwischen Client und BTnode

L2CAP wird für die Kommunikation verwendet. Die Kommunikation verläuft damit paketorientert. Aufgrund einer fehlenden Flusskontrolle auf dem Bluetooth-Modul des BTnode[1] wird diese auf der Applikationsebene nachgebildet: Damit der schnelle Sender (die Workstation oder der PDA) den langsamen Empfänger (den BTnode) nicht mit Paketen überschwemmt, wird jedes Paket mit einer Empfangsbenachrichtigung quittiert. Das Stop-and-Wait-Protokoll bewirkt eine Drosselung der Senderate der Pakete, angepasst an die Verarbeitungszeit der langsamsten Komponente.

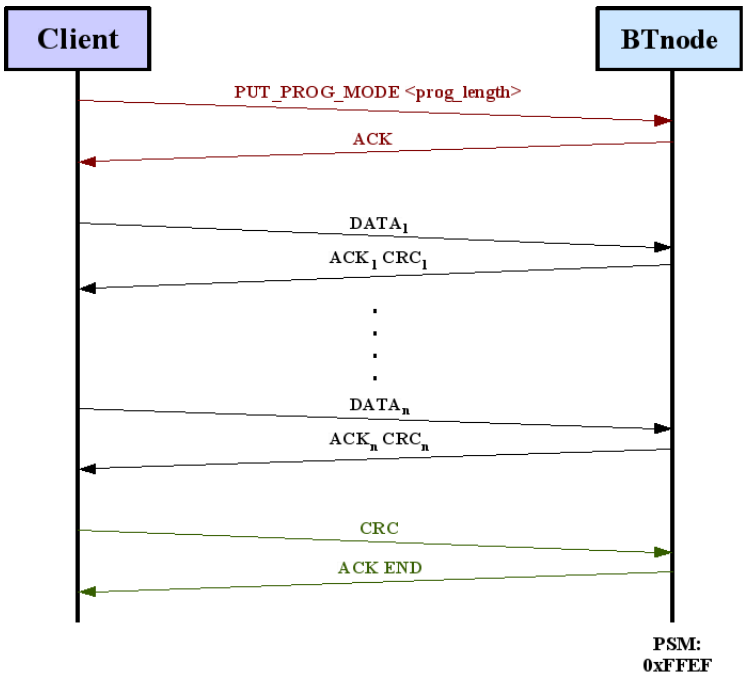


Abbildung 4.1: Das Sequenzdiagramm des RemoteProgramming

Die Kommunikation folgt dem Schema in Abbildung 4.1. Wie man sehen kann, agiert der BTnode auf Kommunikationsebene als Server. Der Client muss sich auf der fixen PSM `0xFFEF` verbinden. Am Anfang wird eine Aufforderung für die Neuprogrammierung mit der Längenangabe `prog_len` gesendet. Nachdem die Länge dem BTnode bekannt ist, werden die Flash-Daten `data` über einzelne Datenpakete DATA_i gesendet. Die Grösse des Datenpakets DATA_i wird durch die maximale Sendeeinheit (MTU) des L2CAP-Kanals begrenzt. Jedes gesendete Paket wird quittiert. Ein allfälliges Problem wird mit NACK signalisiert. Beim Datenpaket DATA_i wird mit der Checksumme CRC_i quittiert. Am Schluss wird der `CRC` vom Client mit den Daten auf dem BTnode

[1] http://lists.ee.ethz.ch/btnode-development/msg00852.html

im SRAM verglichen und bei Übereinstimmung mit END quittiert. Damit ist garantiert, dass die Neuprogrammierung erfolgreich war und der BTnode mit dem neuen Programmcode gestartet werden kann.

Nach dem heutigen Stand (Oktober 2005) wird mit diesem Protokoll eine Durchsatzrate von 1.5 Kilobyte pro Sekunde erreicht, was gegenüber der maximalen Durchsatzrate von 10 Kilobyte pro Sekunde sehr langsam ist. Grund dafür ist die Wartezeit, welche im Stop-and-Wait-Protokoll eine nicht vernachlässigbare Rolle spielt und den Durchsatz massgeblich reduziert.

4.3 Der Serverdienst für den BTnode

Eine C-Implementierung des RemoteProgramming für den BTnode ist bereits im BTnut integriert und eine Demoapplikation ist unter `app/bt-remoteprog` verfügbar. Der Serverdienst wird mit

```
#include <support/bt_remoteprog.h>
:
bt_remoteprog_init(l2cap_stack, rp_reset_cb);
```

gestartet und auf die PSM `0xFFEF` zugeteilt. Zwei Funktionsparameter müssen übergeben werden: ein Pointer auf einen (bereits initialisierten) L2CAP-Stack und ein Funktionspointer. Die Funktion mit dem Rückgabewert, ob der BTnode automatisch neu gestartet werden soll. Der Funktionspointer referenziert auf eine Funktion mit folgender Struktur:

```
u_char rp_reset_cb(u_char arg)
```

Der Client hat die Möglichkeit, über `SET_PROG_MODE_ARG` den Funktionsparameter `arg` zu spezifizieren. Falls man für den Pointer `NULL` übergibt oder die Funktion nicht 0 zurückgibt, wird der BTnode nach 500 Millisekunden über den Watchdog-Timer neu gestartet, ansonsten muss der BTnode manuell oder über die Applikation neu gestartet werden. Um die Verbindung korrekt beenden zu können, ist eine gewisse Wartezeit bis zum Neustart angebracht.

Auf den Quellcode werde ich nicht weiter eingehen. Abbildung 4.2 auf Seite 26 zeigt das Statediagramm des Serverdienstes: Dicke Schrift bezieht sich auf den Typ des empfangenen Pakets und kursive Schrift deutet auf die Funktion hin, in welcher das Paket abgearbeitet wird.

4.4 Der Java-Client

Der Client, welcher auf einer Workstation oder auf einem PDA betrieben werden kann, ist in Java programmiert und besteht im Wesentlichen aus zwei

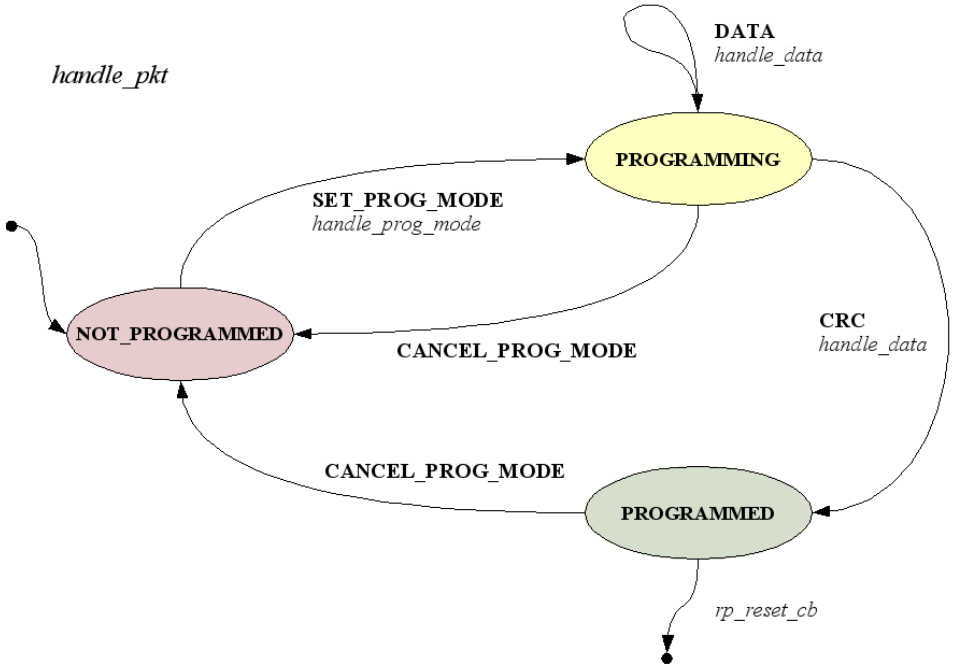


Abbildung 4.2: Das Statediagramm des Serverdienstes RemoteProgramming

Packages:

`ch.ethz.remoteprogrammer`: Dieses Package beinhaltet nicht-grafische Komponenten für das RemoteProgramming auf einem Client und stellt ein simples API für eigene Applikationen zur Verfügung.

- `RemoteProgrammer`: Das Kernmodul, mit welchem der Client einen BTnode ohne grossen Aufwand neu programmieren kann. Es beinhaltet das Protokoll für die Kommunikation und Neuprogrammierung des BTnodes.

- `RemoteProgrammerListener`: `RemoteProgrammerListener` ist das Interface des `RemoteProgrammer` für Fortschrittsbenachrichtigungen.

- `RProg`: `RProg` ist das Frontend für die Kommandozeile. `RProg` greift auf das Kernmodul `RemoteProgrammer` zu und implementiert den Listener `RemoteProgrammerListener`. Zeigt nützliche Informationen wie den prozentualen Fortschritt oder die Datenrate an.

`ch.ethz.btnodeinspector.programming`: Ein AWT-GUI des RemoteProgrammer, welches für den PDA angepasst wurde und auf dem Package `ch.ethz.remoteprogrammer` aufbaut.

- `Frame_Programming`: `Frame_Programming` stellt die grafische Oberfläche bereit.

- `Thread_Programming`: `Thread_Programming` aktualisiert die grafische Oberfläche `Frame_Programming`. `Thread_Programming` greift auf das Kernmodul `RemoteProgrammer` zu und implementiert den `RemoteProgrammerListener` für Fortschrittsmeldungen.

Man hat damit drei Möglichkeiten, einen BTnode über einen Client neu zu programmieren: der BTnodeInspector mit grafischem Frontend, `RProg` mit dem Kommandozeilen-Frontend oder die Einbindung des Kernmoduls `RemoteProgrammer` für den Einsatz in einer eigenen Applikation.

Für `RProg` kann mit

```
java ch.ethz.remoteprogrammer.RProg --help
```

oder

```
# RProg is Main-Class in RemoteProgrammer.jar
java -jar RemoteProgrammer.jar --help
```

eine Hilfe für die Handhabung aufgerufen werden. Für das Kernmodul `RemoteProgrammer` ist im Anhang auf Seite 45 die Javadoc zum Package `ch.ethz.remoteprogrammer` zu finden. Im Konstruktor können auch mehrere BTnodes angegeben werden, welche aber nur sequentiell verarbeitet werden. Für eine parallele Programmierung der BTnodes muss das Package noch erweitert werden. Mehr dazu ist im Ausblick unter Kapitel 7.1 auf Seite 39 zu lesen.

Für die Übermittlung des Programmcodes werden die Daten aus der Datei mit der Endung `.hex` entnommen. Die Daten in dieser Datei sind im Intel-Hex-Format[2] abgelegt. Um diese Daten in das Block-Format des Bootloaders - wie im Kapitel 4.1 auf Seite 23 beschrieben - zu transformieren, wird die Bibliothek `ch.ethz.util.IntelHexFormat` benutzt. Die Daten im ASCII-Format werden in binäre Form umgewandelt, unnötige Zeichen werden weggelassen und um die Gesamtgrösse der Header zu reduzieren, werden grösstmögliche Blöcke gebildet. Damit wird die Datengrösse für die Übertragung um mindestens 50 % reduziert.

Noch ein wichtiger Hinweis: Für den BTnodeInspector müssen die `hex`-Dateien im Verzeichnis `/local` vorliegen, damit sie von ihm gefunden werden. In Abbildung 4.3 auf Seite 28 werden einige Bildschirmfotos von der Oberfläche gezeigt.

[2] http://www.cs.net/lucid/intel.htm

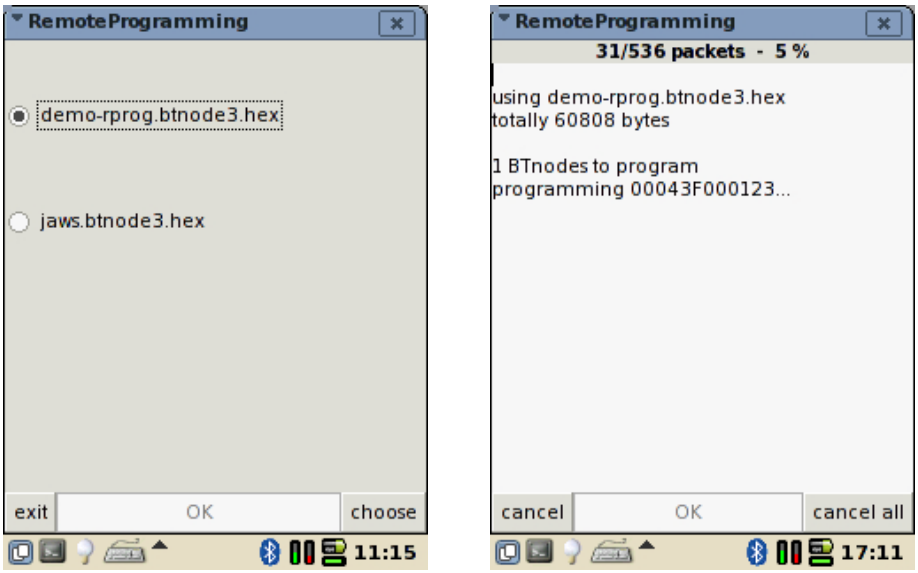


Abbildung 4.3: Bildschirmfotos vom RemoteProgramming: links bei der Auswahl des Programmcodes, rechts mitten in der Programmierphase

Kapitel 5

AttributeRetrieving

JAWS ist leider nicht geeignet, um Informationen von einem Knoten abzufragen. Einerseits können noch keinen Daten von JAWS über L2CAP übermittelt werden, anderseits ist die Abfrage statisch, d.h. die Abfrage muss im vornherein dem Anwender oder Programmierer bekannt sein. In diesem Kapitel wird eine Lösung vorgestellt, die eine einfache Architektur verwendet, um Kurzinformationen bzw. Attribute abzufragen, ohne irgendwelche Kenntnisse über die implementierten Attribute auf einem Knoten vorauszusetzen. Damit bietet der BTnode einen neuen Service mit dem Namen AttributeRetrieving an. In dieser Arbeit werden zwei Attribute zur Verfügung gestellt: die Batteriespannung und der freie Heap-Speicher.

5.1 Eigenschaften von Attributen

Zustandswerte, welche mit einem einzelnen Knoten (hier ein BTnode) verknüpft sind, bezeichnet man als Attribute. Die Werte können konstant sein oder sich über die Zeit verändern. Die Attribute werden beim Kompilieren dynamisch eingebunden und können zur Laufzeit abgefragt werden.
In dieser Arbeit werden die Attribute jeweils einzeln abgefragt. Eine Pull-Technik ist hier angebracht: die Anfragen werden vom Benutzer ausgelöst und sind eher aperiodischer bzw. sporadischer Natur. Eine Aufforderung für ein Attribut wird an den BTnode gesendet und man bekommt die Antwort mit dem Attributwert als Inhalt zurück. Damit der Benutzer weiss, welche Attribute er abfragen kann bzw. will, wird auf dem BTnode ein Inhaltsverzeichnis für Attribute zur Verfügung gestellt. Die Beschreibung der Attribute wird darin in menschen-lesbarer Form als Text für den Anwender abgelegt. So muss kein Wissen über die vorhandenen Attribute vorrausgesetzt werden.

Im Laufe meiner Arbeit unterteilten wir die Attribute in folgende Bereiche, mit einigen Beispielen ergänzt:

Funktion Beschreibung der Funktion des Knoten

- Messungen
- Weiterleitung

Ressourcen Hardwarekomponenten und spezifische Informationen über die Betriebssoftware

- Prozessortaktung
- verfügbarer Speicher
- Anzahl laufender Threads

Energie Informationen über die Energieversorgung

- mittlerer Energieverbrauch
- Batteriespannung
- Zustand der Energiequelle

Lokalität Informationen über die Position des Knoten

- Position
- Mobilität

Kommunikation Informationen über Kommunikationselemente

- Baudrate (Bluetooth/Serial-Port)
- Verbindungsdetails

Umgebungssensoren Informationen über die Umgebung des Knoten

- Temperatursensor
- Magnetfeldsensoren
- Mikrofon

Aktoren Umgebungsagierende Komponenten des Knoten

- LED
- Lautsprecher

Applikationsspezifisch Informationen mit Bezug auf die Applikation

- JAWS (als Beispiel)
  - Baumtopologie

Diese Unterteilung kann hilfreich sein, weil sich die Implementierungen der Attribute innerhalb eines Bereichs nicht gross unterscheiden sollten.

5.2 Die Implementierung eines Attributs

Die Sub-Informationen zu den Attributen werden in einer Metadaten-Struktur abgelegt. Anstatt direkt mit Funktionen, welche einen spezifischen Attributwert zurückgeben, zu arbeiten, wird generell auf die Metadaten referenziert. Die Metadaten zu einem Attribut haben folgende Struktur:

```
struct attr_metadata {
    struct attr_metadata *next;
    u_char * (*function) (u_char *length);
    void (*enable) (void);
    void (*disable) (void);
    u_char result_length;
    u_short gid;
    u_char *desc;
};
```

Das Feld `next` ist für den Aufbau einer einfach verketteten Liste mit weiteren Attribut-Metadaten reserviert. `function` ist ein Funktionspointer, welcher die Adresse zum Resultat zurückgibt. Die maximale Länge des Resultates wird mit `result_length` angegeben und reserviert beim Aufstarten die entsprechende Menge an Speicher. Die Adresse zum Resultat bleibt damit immer konstant. Die genaue Länge des Resultates wird beim Funktionsaufruf mit `length` zurückgegeben. Eine Beschreibung zum Attribut wird über `desc` mit einem String spezifiziert. Da eine Stringoperation für die Identifizierung des Attribut ungeeignet resp. die Verarbeitung bei langen Strings langsam ist, wird mit einem globalen Identifikator `gid` über zwei Bytes gearbeitet (`0x0000` bis `0xFFFF`). Damit sind $2^{16} = 65'536$ Identifikatoren möglich. Aber man muss bei der Definitionen der Metadaten selber besorgt sein, dass der globale Identifikator `gid` eindeutig bleibt. Um z.B. Strom zu sparen, können die Attribute temporär deaktiviert werden. Es werden damit zwei weitere Funktionspointer, `enable` und `disable`, angegeben, welche - falls nicht mit `NULL` definiert - beim Aktivieren oder Deaktivieren der Attribute ausgeführt werden. Damit kann die Steuerung der Aktivierung bzw. Deaktivierung auch in andere Threads auslagert werden, z.B. für das Power-Management.

Als Beispiele wurden Attribute zur Batteriespannung und zur Angabe des frei-verfügbaren Heap-Speicher implementiert und sind im Anhang auf Seite 45 unter `attr_battery.c` und `attr_heap.c` zu finden.

5.3 Der Serverdienst für den BTnode

Abbildung 5.1 zeigt einen Überblick über den Aufbau des Serverdienstes. `attr` dient als Verwaltungsstelle für Attribute und zur Interpretation der Metadaten. Die Metadaten werden hier registriert, über das Feld `next` einfach verkettet und zusätzlich in einer Lookup-Tabelle abgelegt. Die Lookup-Tabelle bietet

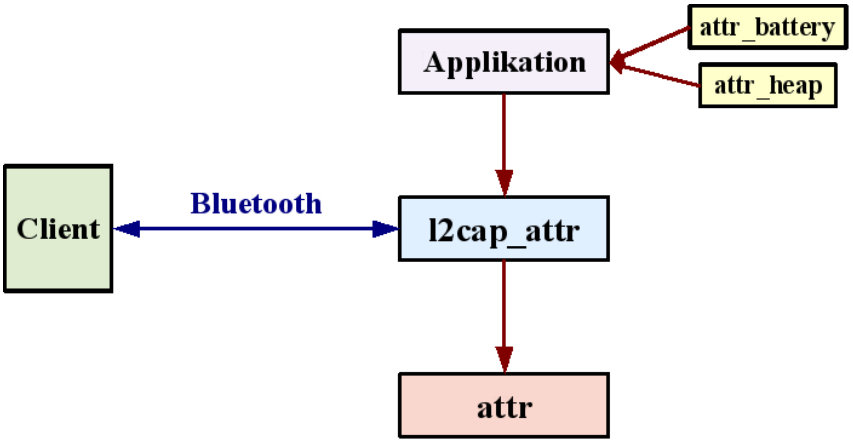


Abbildung 5.1: Das Blockschema des Serverdienstes AttributeRetrieving

die Möglichkeit, direkt und damit schneller auf die Metadaten resp. auf die Funktionen zugreifen zu können. Die Lookup-Tabelle ist für 255 Attribute dimensioniert. Mit dem Index der Lookup-Tabelle ist auch ein interner, lokaler Identifikator `id` mit dem Addressbereich zwischen `0x01` und `0xFF` verfügbar, welcher alternativ zum globalen Identifikator `gid` beim Aufruf eines Attributes angegeben werden kann. Das Null-Byte `0x00` ist für die Abfrage des Verzeichnisses reserviert. `attr` ist aber weder für den Transport der Attribute noch für den Aufbau und das Bereitstellen des Verzeichnisses zuständig. Es werden aber Hilfsfunktionen mitgeliefert, die zum Bau des Verzeichnisses nützlich sind. Im Anhang auf Seite 45 ist eine Beschreibung der Funktionen zu finden.
`l2cap_attr` übernimmt folgende Aufgaben: Der Aufbau und Unterhalt des Verzeichnisses und die Möglichkeit der Attribut-Abfrage über L2CAP. Ein L2CAP-Paket mit einem Null-Byte (`0x00`) als Inhalt muss zur PSM `0xE001` gesendet werden, um das Verzeichnis abzufragen. Das Verzeichnis wird bei jeder Abfrage on-the-fly erzeugt. Ein Verzeichnis beinhaltet mehrere Einträge mit folgender Struktur:

```
id      gid      psm      desc      '\n'
 1       2        2   string_length   1
```

Damit wird auch die PSM für den Zugriff auf das Attribut mitgegeben. Die Liste wird schliesslich mit `'\0'` terminiert.
Die Einbindung in eine Applikation findet auf folgende Weise statt:

```
#include "l2cap_attr.h"
#include "attr_battery.h"
#include "attr_heap.h"
:
// register attributes
l2cap_attr_register(&battery_metadata);
l2cap_attr_register(&heap_metadata);
// initialise attribute retrieving service over l2cap
l2cap_attr_init(l2cap_stack);
```

Zuerst werden die Attribute über die Metadaten registriert. Erst nach der Registrierung können mit `l2cap_attr_init` der Service initialisiert und die Attribute über L2CAP angeboten werden.
Der Client kann nun ein L2CAP-Paket mit dem Identifikator `id` bzw. `gid` als Inhalt an den BTnode schicken und bekommt ein Paket mit folgendem Inhalt zurück:

```
status      payload
   1     string_length
```

Falls das Statusbyte `0x00` ergibt, war die Abfrage erfolgreich und der Attributwert wird im `payload`-Feld mitgeliefert.

5.4 Der Java-Client

Der Client, welcher auf einer Workstation oder auf einem PDA betrieben werden kann, ist in Java programmiert und besteht aus drei Klassen, welche im Package `ch.ethz.btnodeinspector.attribute` bereitliegen:

- `Frame_Attribute`: `Frame_Attribute` stellt die grafische Oberfläche bereit. Die Oberfläche besteht aus einem Informationsfeld mit Bluetooth-Eigenschaften des BTnode, einem Auswahlfenster mit der Auflistung der Attribute und einem Informationsfeld über die Signalstärke der Bluetooth-Verbindung.
- `Thread_Attribute`: `Thread_Attribute` aktualisiert `Frame_Topology`: Holt am Anfang das Verzeichnis vom BTnode und baut das Auswahlfenster auf, nach dem Aufbau der Verbindung aktualisiert es kontinuierlich den Attributwert und den Wert der Signalstärke.
- `AttributeListEntry`: Ein Eintrag der Attributliste, bestehend aus `id` und `psm`.

`Thread_Topology` übernimmt die L2CAP-Kommunikation mit dem BTnode. Im Client werden zudem, damit das Fenster nicht allzu leer erscheint, einige Zusatzinformationen gezeigt: die Bluetooth-Eigenschaften des BTnode und die Signalstärke der Verbindung. Die Bluetooth-Eigenschaften wie Name, Hardware-Adresse und Klasse werden über einen Inquiry offengelegt. Die Signalstärke wird über eine AvetanaBT-Klasse geliefert und ist nicht JSR-82-konform. Im JSR-82 wird leider keine Methode spezifiziert, die die Information der Signalstärke anbietet.
Abbildung 5.2 auf Seite 34 zeigt einige Bildschirmfotos von der Oberfläche.

Abbildung 5.2: Bildschirmfotos vom AttributeRetrieving: links wird die Attributliste gezeigt, rechts wird das Attribut 'Battery Voltage' angewählt

Kapitel 6

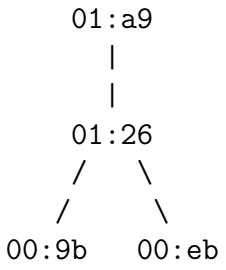
TopologyDrawing

Mit JAWS können BTnodes untereinander selbständig ein Netzwerk unterhalten. JAWS baut für die Multihop-Kommunikation eine Baumstruktur auf. Interessant in diesem Zusammenhang ist die Information wie die Pakete das Netz durchlaufen. In diesem Kapitel wird eine Lösung vorgestellt, die Topologie in Form eines Baums auf dem PDA darstellen zu können.

6.1 Nützliche Kommandos in JAWS

JAWS unterhält eine Kommando-Shell über den seriellen Port, womit Befehle zur Ausführung übermittelt werden. Im Anhang auf Seite 45 sind die Kommandos für JAWS aufgelistet. Es gibt ein Kommando, das für unsere Anwendung nützlich ist. Mit der Ausgabe von '`tp trace`' kann der Baum mit einem einfachen Algorithmus generiert werden. Dieses Kommando sorgt dafür, dass, mit Ausnahme der Wurzel, jeder Knoten im Netz seine eigene Adresse im Paket mitführt. Das Paket wird einerseits zur Wurzel zurückgesendet, andererseits an die Nachbarn weitergeleitet.

Angenommen der Baum hat diese Struktur:



Dann wird folgende Ausgabe auf der Wurzel bzw. auf dem BTnode mit der Adresse `00:a9` ausgegeben:

```
[jaws@00:a9]$ tp trace
```

```
Trace result:
1. 00:04:3f:00:01:26

Trace result:
1. 00:04:3f:00:01:26
2. 00:04:3f:00:00:9b

Trace result:
1. 00:04:3f:00:01:26
2. 00:04:3f:00:00:eb
```

Mit der Ausgabe von '`tp trace`' können die Kanten und Distanzen von der Wurzel ausgelesen werden. Der Baum kann mit diesen Daten auf einfachste Weise aufgebaut werden.

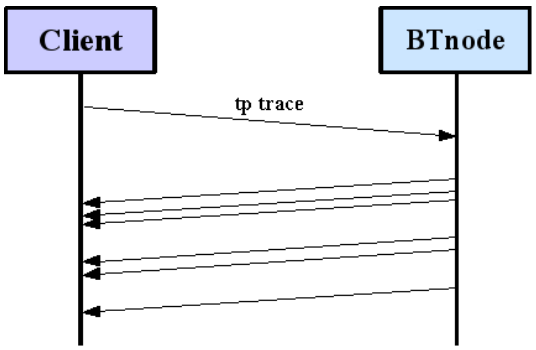


Abbildung 6.1: Das Sequenzdiagramm einer 'tp trace'-Abfrage

Pro Knoten im Netz, dessen Grösse unbekannt ist, wird eine Rückgabe generiert. Darum wird ein Timeout für die Kommunikation benötigt, da die Rückgabe, wie der Abbildung 6.1 anzumerken ist, nicht-deterministisch verläuft.

JAWS wird normalerweise über die serielle Anbindung bedient und gesteuert. JAWS bietet aber auch eine Schnittstelle an, um Kommandos über L2CAP aufnehmen zu können (sogenannte Remote Commands). Ein grosses Problem der Remote Commands ist aber, dass sie keine Rückgabewerte liefern können. Sie werden nur auf dem seriellen Port ausgegeben. Durch einen Quick&Dirty-Hack ist es möglich, die benötigte Rückgabe zugleich über L2CAP auf die verbundenen Geräte weiterzuleiten.

Mit '`dsn cmd 0 blink`' wird ein Broadcast-Kommando zum Blinken an alle Knoten übermittelt. Dieses Kommando gibt keine notwendigen bzw. nützlichen Daten zurück, ein Hack war somit nicht nötig.

6.2 Der Java-Client

Der Client, welcher auf einer Workstation oder auf einem PDA betrieben werden kann, ist in Java programmiert. Der Client bietet mit Blinken und der Darstellung der Topologie zwei Funktionalitäten und besteht aus insgesamt fünf Klassen, welche im Package `ch.ethz.btnodeinspector.topology` bereitliegen:

- `Frame_Topology`: `Frame_Topology` stellt die grafische Oberfläche bereit, bestehend aus einem Nachrichtenfeld, einem Zeichnungsfeld und zwei Buttons.
- `Thread_Topology`: `Thread_Topology` aktualisiert `Frame_Topology`: Sendet Kommandos zum JAWS-Knoten beim Betätigen der Buttons, empfängt und interpretiert die Rückgabe und benachrichtigt `Frame_Topology` über die Kanten.
- `Canvas_Topology`: Das Zeichnungsfeld im `Frame_Topology` kann aus den Informationen der Objekte `NodeEdge` und `NodePosition` den Baum zeichnen.
- `NodeEdge`: `NodeEdge` umfasst eine Kante des Baums und die Distanz zur Wurzel.
- `NodePosition`: `NodePosition` umfasst die Positionen eines Knoten im Zeichnungsfeld `Canvas_Topology`.

`Thread_Topology` übernimmt die L2CAP-Kommunikation mit dem JAWS-Knoten: Der Thread sendet Kommandos, empfängt und interpretiert die Antworten. Im Falle, dass eine neue Kante gefunden wurde, wird `Canvas_Topology` darüber informiert und aktualisiert sich dementsprechend. Somit wird der Baum nach jeder neuen Kante neu gezeichnet. Für die Positionierung der Knoten werden Informationen wie maximale Distanz und maximale Anzahl von Knoten auf jeweils gleicher Distanz benötigt. Die maximale Distanz wird für die vertikale Aufteilung, die maximale Anzahl Knoten für die horizontale Aufteilung verwendet. Abbildung 6.2 auf Seite 38 zeigt einige Bildschirmfotos vom Aufbau des Baumes und der Positionierung der Knoten.

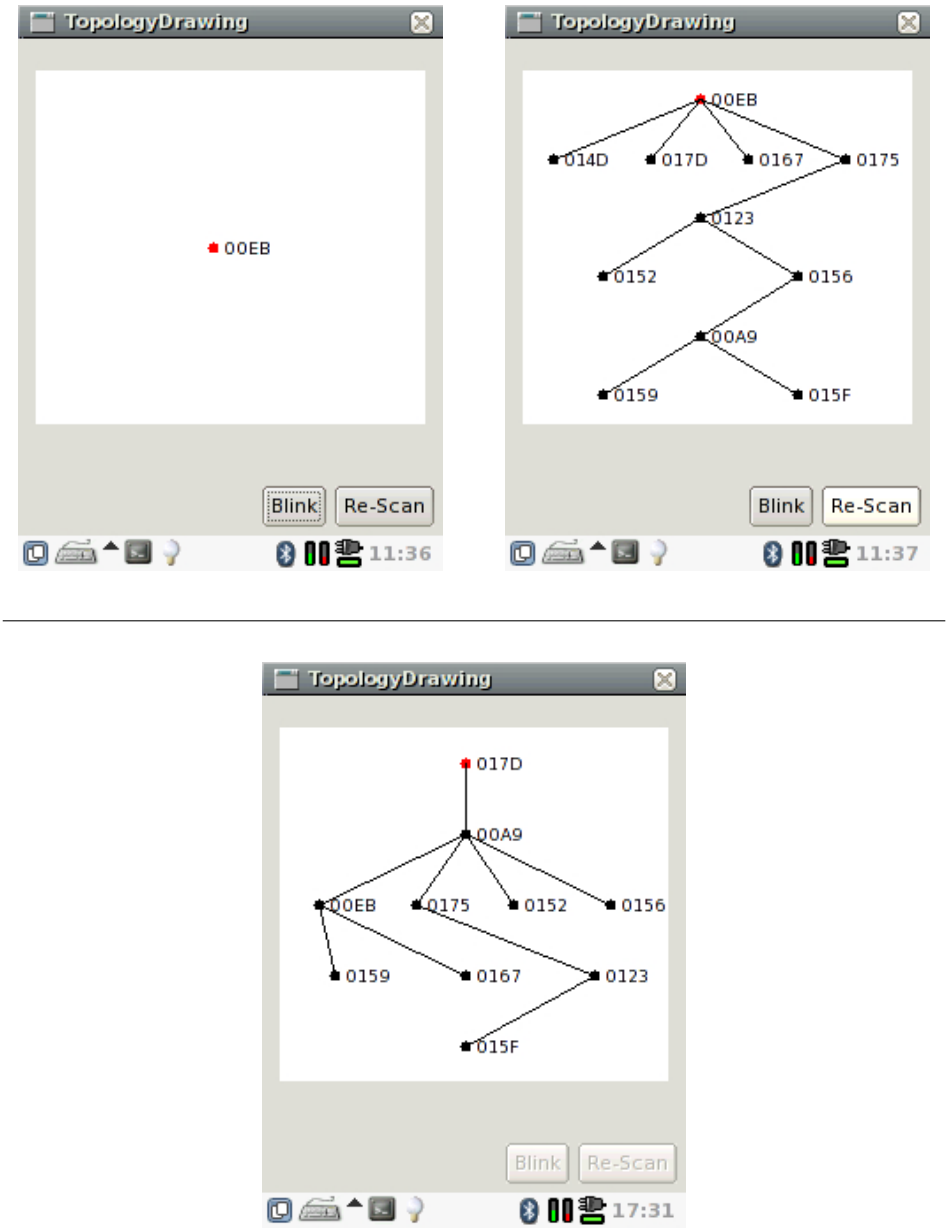


Abbildung 6.2: Bildschirmfotos vom TopologyDrawing: Oben wurde der Button 'Re-Scan' für die Darstellung der Topologie angeklickt, unten wird eine andere Topologie aufgezeigt

Kapitel 7

Ausblick

Die Arbeit wurde gemäss der Aufgabenbeschreibung erfolgreich ausgeführt. In diesem Kapitel werden Punkte vorgeschlagen, welche für die Fortführung der Arbeit von Nutzen sein können. Es wird auch hingewiesen auf einige Verbesserungsvorschläge zu bestehenden Problemen, die im Verlauf der Arbeit aufgetreten sind und noch nicht weiter behandelt wurden.

7.1 RemoteProgramming

Der Durchsatz von 1.5 Kilobytes pro Sekunde ist meiner Meinung nach zu langsam und kann noch optimiert werden. Eine Möglichkeit für die Optimierung wäre, die BTnodes anstatt sequentiell parallel zu programmieren. In einem Piconetz können bis zu sieben Verbindungen simultan unterhalten werden. Damit kann der Durchsatz auf dem Client theoretisch auf maximal $7 * 1.5KB/s = 10.5KB/s$ erhöht werden. Eine weitere Möglichkeit besteht darin, die Kommunikation auf dem Sliding-Window-Protokoll[1] zu bauen, um damit den Nachteil mit der Wartezeit beim Stop-and-Wait-Protokoll zu beheben. Bei funktionaler Flusskontrolle des Bluetooth-Moduls auf dem BTnode kann auch gänzlich auf eine Flusskontrolle in der Applikation verzichtet werden. Zu diesem Zweck ist in der Software von Client und BTnode ein "Schalter" eingebaut, um die Bestätigungen auszuschalten. Erwartungsgemäss kann mit den letzten beiden Lösungsvorschlägen ein Durchsatz gegen 10 Kilobytes pro Sekunde auf dem Server bzw. BTnode erreicht werden. Zusammen mit dem ersten Vorschlag erreicht man also auf dem Client ein Durchsatz von bis zu 70 Kilobytes pro Sekunde.
Ein weiteres Problem: Die Übertragung auf dem PDA ist etwa halb so schnell wie die auf einer Workstation (z.B. auf meinem Laptop). Den Grund dafür habe ich nicht finden können: Der Prozessor des PDA ist nicht ausgelastet, Speicher ist genug vorhanden, das Bluetooth-Modul auf dem PDA bietet einen Durchsatz von 60 Kilobytes pro Sekunde an und die Linux-Softwarekomponenten sind für PDA und Workstation grösstenteils dieselben. Hier ist eine nähere Be-

[1] http://en.wikipedia.org/wiki/Sliding_window

trachtung notwendig.

7.2 AttributeRetrieving

Zurzeit läuft der ganze Kommunikationsablauf im Java Client in einem Thread ab, welcher noch im BTnodeInspector eingebettet ist. Eine Trennung durch ein Modul mit einer definierten API ist sinnvoll, wie es beim RemoteProgrammer gemacht wurde. Damit kann einerseits wieder ein Kommandozeilen-Frontend angeboten werden, andererseits wird die Einbindung in andere Anwendungen vereinfacht.
Zur Implementierung von periodischen Abfragen kann die Push-Technik angewendet werden: Der Client meldet sich für die Attribute beim BTnode an (subscription). Danach sendet der BTnode in konstanten Intervallen die Attributwerte an den Client zurück. Das geht solange, bis sich der Client abmeldet oder die Verbindung abbricht (unsubscription).

7.3 TopologyDrawing

Die Quick&Dirty-Hacks sind langfristig sicher keine Lösung. JAWS sollte für eine sinnvolle Unterstützung der Remote Commands über L2CAP auf bidirektionale Kommunikation erweitert werden.
Ein weiteres Problem: Das Kommando '`tp trace`' ist aufgrund von Instabilitäten während der Ausführung in grösseren Netzen nicht geeignet. Entweder sollte das Kommando besser implementiert oder ganz aus der Kommando-Referenz gestrichen werden. Im letzteren Fall kann für die Konstruktion des Baumes ein anderer Algorithmus angewendet werden, welcher rekursiv die Verbindungstabellen aller BTnodes abfrägt. In der Semesterarbeit von Kevin Martin[2] wird ein solcher Algorithmus angewendet.

7.4 BTnodeInspector

Sobald das SDP-Protokoll für BTnut fertig implementiert wird, kann für RemoteProgramming, AttributeRetrieving und JAWS jeweils ein eigener Service-Eintrag generiert werden. Damit ist eine Suchanfrage wie z.B. "alle BTnodes sollen sich melden, welche RemoteProgramming anbieten können" möglich und kann als Filter im BTnodeInspector eingesetzt werden.

[2]Online Sensor-Network Monitoring, SA-2005-26, http://www.tik.ee.ethz.ch/∼dyer

7.5 L2CAPCommunication

Für die Implementierung des Timeouts wird zurzeit gepollt. Es gibt aber auch andere, effizientere Möglichkeiten um Timeouts zu implementieren. `L2CAP-Communication` kann dementsprechend erweitert werden.

Literaturverzeichnis

[1] **B. Kumar, P. Kline, T. Thompson**, "Bluetooth Application Programming with the Java APIs", 2003, Morgan Kaufmann, ISBN 1-55860-934-2

[2] **B. Hopkins, R. Antony**, "Bluetooth for Java", 2003, Springer-Verlag, ISBN 1-59059-078-3

[3] **Guido Krüger**, "Handbuch der Java-Programmierung", Studentenausgabe, 3. Auflage, Addison-Wesley, ISBN 3-8273-2120-4

[4] **G. Vaughan, B. Elliston, T. Tromey, I. Taylor**, "GNU Autoconf, Automake and Libtool", 2000, New Riders, ISBN 1-57870-190-2

[5] **K. Römer, F. Mattern**, "Event-Based Systems for Detecting Real-World States with Sensor Networks: A Critical Analysis", DEST Workshop on Signal Processing in Sensor Networks at ISSNIP, S. 389-395, Melbourne, Australien, Dezember 2004

[6] **K. Römer, F. Mattern**, "The Design Space of Wireless Sensor Networks", IEEE Wireless Communications, Vol. 11, No. 6, S. 54-61, Dezember 2004

[7] **J. Beutel, M. Dyer, L. Meier, M. Ringwald, L. Thiele**, "Next-Generation Deployment Support for Sensor Networks", TIK Report No. 207, ETH Zürich, Schweiz, November 2004

[8] **Pav Lucistnik**, "Das FreeBSD-Handbuch", Kapitel 26.4. Bluetooth, http://www.freebsd.org/doc/de_DE.ISO8859-1/books/handbook/network-bluetooth.html

Anhang A

OpenEmbedded: avetanabt_cvs.bb

OpenEmbedded: fbvncserver_0.9.4.bb

OpenEmbedded: subversion_1.2.0.bb

Bash-Skript: backup.sh

Javadoc: RemoteProgrammer

BTnut: attr_battery.c

BTnut: attr_heap.c

JAWS: Kommandos

Quellcode-Repositories

A Anhang

```
#
# OpenEmbedded: avetanabt_cvs.bb
#

DESCRIPTION = "avetanaBT: Bluetooth API implementation for Java (JSR-82)"
SECTION = "devel"
DEPENDS = "findutils-native jikes-native kaffeh-native fastjar-native bluez-libs
          classpath"
MAINTAINER = "Mustafa Yuecel <yuecelm@ee.ethz.ch>"
LICENSE = "GPL"
HOMEPAGE = "http://sourceforge.net/projects/avetanabt/"

PV = "0.0cvs${CVSDATE}"
PR = "r3"

SRC_URI = "cvs://anonymous@cvs.sourceforge.net/cvsroot/avetanabt;
          module=avetanabt \
          file://maxConnectedDevices.patch;patch=1"

S = "${WORKDIR}/avetanabt"

PACKAGES = "${PN}"
FILES_${PN} = "${libdir}/libavetanaBT.so ${datadir}/avetanabt/avetanaBT.jar"

do_compile() {

  # doing nearly the same as in Makefile written...

  # clean build directory
  ${STAGING_BINDIR}/mkdir -p build
  ${STAGING_BINDIR}/rm -fr build/*

  # generate classes
  # javac -> jikes
  ${STAGING_BINDIR}/find {de,javax,com} -iname *.java > file.list
  ${STAGING_BINDIR}/jikes -verbose --bootclasspath ${STAGING_DIR}/${BUILD_SYS}
    /share/kaffeh/rt.jar -d build @file.list

  # create own version.xml (add version information available at runtime)
  ${STAGING_BINDIR}/head -n 4 version.xml >> build/version.xml
  ${STAGING_BINDIR}/echo "    <build value=\"cvs${CVSDATE}\" date=\"${CVSDATE}\"
    time=\"${@time.strftime('%H:%M',time.gmtime())}\"/>">> build/version.xml
  ${STAGING_BINDIR}/tail -n 3 version.xml >> build/version.xml

  # move classes into jar archive
  # jar -> fastjar
  ${STAGING_BINDIR}/fastjar -v -cf avetanaBT.jar -C build de -C build javax -C
    build com -C build version.xml

  # JNI generated header file - de_avetana_bluetooth_stack_BlueZ.h
  # javah -> kaffeh
  ${STAGING_BINDIR}/kaffeh -jni -classpath avetanaBT.jar:${STAGING_DIR}/${
    BUILD_SYS}/share/kaffeh/rt.jar -d c de.avetana.bluetooth.stack.BlueZ
```

```
  # Native language (C) library - libavetanaBT.so
  ${CXX} ${CXXFLAGS}  -shared -lbluetooth -I${STAGING_INCDIR}/classpath c/BlueZ.
    cpp -o libavetanaBT.so ${LDFLAGS}

}

do_stage() {

  install -d ${STAGING_DIR}/${BUILD_SYS}/share/avetanabt
  install avetanaBT.jar ${STAGING_DIR}/${BUILD_SYS}/share/avetanabt/

}

do_install() {

  install -d ${D}${libdir}
  install -m 0755 libavetanaBT.so ${D}${libdir}/

  install -d ${D}${datadir}/avetanabt
  install avetanaBT.jar ${D}${datadir}/avetanabt/

}
```

```
#
# OpenEmbedded: fbvncserver_0.9.4.bb
#

DESCRIPTION = "Framebuffer VNC server"
LICENSE = "GPL"
SECTION = "console/utils"
#DEPENDS = "libvncserver jpeg zlib"
# using older version due of error with libvncserver-0.7.1
# fbvncserver.c:577: error: structure has no member named 'rfbAlwaysShared'
# fbvncserver.c:602: error: structure has no member named 'rfbClientHead'
DEPENDS = "libvncserver-0.6 jpeg zlib"
RDEPENDS = "fbvncserver-kmodule libvncserver-storepasswd libvncserver-
          javaapplet"
PR = "r2"

SRC_URI = "http://sdgsystems.com/download/fbvncserver-${PV}.tar.gz \
           file://libvncs0.6.patch;patch=1 \
           file://paths.patch;patch=1 \
           file://kernelinclude.patch;patch=1 \
           file://buildfix.patch;patch=1 \
           file://ipaq.patch;patch=1 \
           file://init"

S = "${WORKDIR}/fbvncserver-${PV}"

export INCLUDES = "-I${STAGING_INCDIR}"

export LIBS = "-L${STAGING_LIBDIR} -lpthread"
export VNCSERVER_DIR = "${STAGING_LIBDIR}"
export ZAURUS_ZLIB_LIBS = "${STAGING_LIBDIR}"
export ZAURUS_JPEG_LIBS = "${STAGING_LIBDIR}"

inherit update-rc.d

INITSCRIPT_NAME = "fbvncinput"
INITSCRIPT_PARAMS = "defaults 97"

FBVNCSERVER_SYSTEM = "zaurus"
FBVNCSERVER_SYSTEM_h3600 = "ipaq"
FBVNCSERVER_SYSTEM_h3900 = "ipaq"

do_compile () {

oe_runmake ${FBVNCSERVER_SYSTEM}_fbvncserver ${FBVNCSERVER_SYSTEM}_tssimd

}

do_install () {

install -d ${D}${bindir}
install -m 0755 ${FBVNCSERVER_SYSTEM}_fbvncserver ${D}${bindir}/fbvncserver
install -m 0755 ${FBVNCSERVER_SYSTEM}_tssimd ${D}${bindir}/tssimd
```

```
install -d ${D}${datadir}/fbvncserver
install -m 0644 ${FBVNCSERVER_SYSTEM}_panel.jpg ${D}${datadir}/fbvncserver/

install -d ${D}${sysconfdir}/init.d
install -m 0755 ${WORKDIR}/init ${D}${sysconfdir}/init.d/fbvncinput

}

FILES_${PN} += " /usr/share/fbvncserver/*.jpg"
```

```
#
# OpenEmbedded: subversion_1.2.0.bb
#

DESCRIPTION = "The Subversion (svn) client"
SECTION = "console/network"
DEPENDS = "apr-util"
MAINTAINER = "Mustafa Yuecel <yuecelm@ee.ethz.ch>"
LICENSE = "GPL"

PR = "r0"

SRC_URI = "http://subversion.tigris.org/downloads/${P}.tar.bz2 \
    file://disable-revision-install.patch;patch=1"

EXTRA_OECONF = "--without-neon --without-berkeley-db --without-apxs \
                --without-apache --without-swig \
                --with-apr=${STAGING_BINDIR} \
                --with-apr-util=${STAGING_BINDIR}"

inherit autotools

do_configure() {

  oe_runconf

}
```

```
#
# Bash-Skript: backup.sh
#

#!/bin/sh

echo "Whole procedure takes ~45min.  Have patience."
/sbin/modprobe mtdchar > /dev/null
echo "backup root..."
dd if=/dev/mtd/1ro of=/media/card/image/imagefile.jffs2 > /dev/null
/sbin/modprobe loop > /dev/null
/sbin/losetup /dev/loop/0 /media/card/image/imagefile.jffs2
/sbin/modprobe blkmtd-24 erasesz=256 device=/dev/loop/0 > /dev/null
echo "mount root-image..."
mount -t jffs2 /dev/mtdblock/4 /mnt/image/
echo "clean root-image..."
mkfs.jffs2 --root=/mnt/image --eraseblock=262144 --pad --
                  output=/media/card/image/image-root.jffs2 > /dev/null
rm /media/card/image/imagefile.jffs2
umount /mnt/image
/sbin/rmmod blkmtd-24
/sbin/losetup -d /dev/loop/0
echo "backup local..."
dd if=/dev/mtd/2ro of=/media/card/image/imagefile.jffs2 > /dev/null
/sbin/losetup /dev/loop/0 /media/card/image/imagefile.jffs2
/sbin/modprobe blkmtd-24 erasesz=256 device=/dev/loop/0 > /dev/null
echo "mount local-image..."
mount -t jffs2 /dev/mtdblock/4 /mnt/image/
echo "clean local-image..."
mkfs.jffs2 --root=/mnt/image --eraseblock=262144 --pad --
                  output=/media/card/image/image-local.jffs2 > /dev/null
rm /media/card/image/imagefile.jffs2
umount /mnt/image
/sbin/rmmod blkmtd-24
/sbin/losetup -d /dev/loop/0
/sbin/rmmod loop
/sbin/rmmod mtdchar
```

Package
ch.ethz.remoteprogrammer

ch.ethz.remoteprogrammer
Class RemoteProgrammer

```
java.lang.Object
   |
   +-ch.ethz.remoteprogrammer.RemoteProgrammer
```

public class **RemoteProgrammer**
extends java.lang.Object

Module for remote programming device (BTnode running Nut/OS) that supports `bt-remoteprog`

uses `L2CAPCommunication`

Constructor Summary

public	`RemoteProgrammer()` Constructor
public	`RemoteProgrammer(boolean debugMode)` Constructor
public	`RemoteProgrammer(RemoteProgrammerListener listener)` Constructor
public	`RemoteProgrammer(RemoteProgrammerListener listener, boolean debugMode)` Constructor

Method Summary

void	`doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn)` Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog` use before `setFile(file)`
void	`doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn, java.io.File fp)` Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog`
void	`doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn, java.io.File fp, int len)` Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog`
void	`doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn, int len)` Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog` use before `setFile(file)`
static int	`getDefaultLength()` Returns default payload length of data-packet
static java.lang.String	`getPSM()` Returns PSM as hexstring
void	`interrupt()` Interrupts remote programming process
void	`setFile(java.io.File fp)` Sets File to program

Methods inherited from class `java.lang.Object`
`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

RemoteProgrammer

`public` **`RemoteProgrammer`**`()`

Constructor

RemoteProgrammer

`public` **`RemoteProgrammer`**`(boolean debugMode)`

Constructor

Parameters:
`debugMode` - enable debug mode

RemoteProgrammer

`public` **`RemoteProgrammer`**`(RemoteProgrammerListener listener)`

Constructor

Parameters:
`listener` - listener for progress updates

RemoteProgrammer

`public` **`RemoteProgrammer`**`(RemoteProgrammerListener listener,`
`boolean debugMode)`

Constructor

Parameters:
`listener` - listener for progress updates
`debugMode` - enable debug mode

Methods

getPSM

`public static java.lang.String` **`getPSM`**`()`

Returns PSM as hexstring

Returns:
psm as hexstring

getDefaultLength

`public static int` **`getDefaultLength`**`()`

(continued on next page)

(continued from last page)

Returns default payload length of data-packet

Returns:
length in bytes

---

setFile

```
public void setFile(java.io.File fp)
  throws java.lang.Exception
```

Sets File to program

Parameters:
`fp` - reference to `File` of IntelHex file to be transmitted

Throws:
`Exception`

---

doProgramming

```
public void doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn)
  throws java.lang.Exception
```

Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog`

use before `setFile(file)`

Parameters:
`conn` - reference to L2CAPConnInstance

Throws:
`Exception`

---

doProgramming

```
public void doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn,
          java.io.File fp)
  throws java.lang.Exception
```

Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog`

Parameters:
`conn` - reference to L2CAPConnInstance
`fp` - reference to `File` of .hex file to be transmitted

Throws:
`Exception`

---

doProgramming

```
public void doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn,
          java.io.File fp,
          int len)
  throws java.lang.Exception
```

Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog`

Parameters:
`conn` - reference to L2CAPConnInstance

(continued on next page)

(continued from last page)

`fp` - reference to `File` of IntelHex file to be transmitted
`len` - maximum length of one data packet

Throws:
`Exception`

---

doProgramming

```
public void doProgramming(ch.ethz.l2cap.L2CAPConnInstance conn,
          int len)
  throws java.lang.Exception
```

Programms remote device (BTnode with Nut/OS) that supports `bt-remoteprog`

use before `setFile(file)`

Parameters:
`conn` - reference to L2CAPConnInstance
`len` - maximum length of one data packet

Throws:
`Exception`

---

interrupt

```
public void interrupt()
```

Interrupts remote programming process

ch.ethz.remoteprogrammer
Interface RemoteProgrammerListener

public interface **RemoteProgrammerListener**
extends

Listener interface for `RemoteProgrammer`

Method Summary

`void`	`updateTotalBytesToTransmit(int num)` Updates total length in bytes to transmit (at beginning)
`void`	`updateTotalPacketsToTransmit(int num)` Updates total packets to transmit (at beginning)
`void`	`updateTransmittedPackets(int num)` Updates transmitted packets (begins with 0, steps while programming, ends with TotalPackets)

Methods

updateTotalBytesToTransmit

`public void updateTotalBytesToTransmit(int num)`

Updates total length in bytes to transmit (at beginning)

Parameters:
`num` - total bytes

updateTotalPacketsToTransmit

`public void updateTotalPacketsToTransmit(int num)`

Updates total packets to transmit (at beginning)

Parameters:
`num` - total packets

updateTransmittedPackets

`public void updateTransmittedPackets(int num)`

Updates transmitted packets (begins with 0, steps while programming, ends with TotalPackets)

Parameters:
`num` - transmitted packets

Package
ch.ethz.util

ch.ethz.util
Class IntelHexFormat

```
java.lang.Object
   |
   +-ch.ethz.util.IntelHexFormat
```

public class **IntelHexFormat**
extends java.lang.Object

Class for handling Intel Hex Format Files. Supports building of greater blocks.

Constructor Summary

public	IntelHexFormat()

Method Summary

static byte[]	IntelHexFormatToByteArray(java.io.File fp) Convert IntelHex file to a formatted byte array
static byte[]	IntelHexFormatToByteArray(java.io.File fp, int maxblocklen) Convert IntelHex file to a formatted byte array
static byte[]	IntToFixedByteArray(int integer, int bytelen) Convert integer to a fixed sized byte array.
static boolean	isIntelHexFormat(java.io.File fp) Check if file is in IntelHex format

Methods inherited from class `java.lang.Object`

`equals, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Constructors

IntelHexFormat

```
public IntelHexFormat()
```

Methods

IntelHexFormatToByteArray

```
public static byte[] IntelHexFormatToByteArray(java.io.File fp)
  throws java.lang.Exception
```

Convert IntelHex file to a formatted byte array

Parameters:

(continued from last page)

`fp` - a `File` instance

Returns:

formatted byte array

Throws:

`Exception`

IntelHexFormatToByteArray

```
public static byte[] IntelHexFormatToByteArray(java.io.File fp,
         int maxblocklen)
  throws java.lang.Exception
```

Convert IntelHex file to a formatted byte array

Parameters:

`fp` - a `File` instance

`maxblocklen` - maximum length of a block

Returns:

formatted byte array

Throws:

`Exception`

IntToFixedByteArray

```
public static byte[] IntToFixedByteArray(int integer,
         int bytelen)
  throws java.lang.Exception
```

Convert integer to a fixed sized byte array. Empty bytes will be filled with 0. Oversized array will be cutted to fit into the byte array.

Parameters:

`integer` - number to convert

`bytelen` - the length of the returned byte array

Returns:

formatted byte array

Throws:

`Exception`

isIntelHexFormat

```
public static boolean isIntelHexFormat(java.io.File fp)
```

Check if file is in IntelHex format

Parameters:

`fp` - a `File` instance

Returns:

true, if file is in IntelHex format

```c
//
// BTnut: attr_battery.c
//

/**
* \date 07.07.2005
*
* \author Mustafa Yuecel <yuecelm@ee.ethz.ch>
*
* \brief attribute retrieving example with battery voltage
*/

#define ID 0x0001
#define DESC "Battery Voltage"
#define RESULT_MAX_LEN 7

#include <string.h>
#include <stdio.h>
#include <hardware/btn-bat.h>
#include "attr.h"
#include "attr_battery.h"

static u_char desc[strlen(DESC) + 1] = DESC;

static u_char result[RESULT_MAX_LEN];

/** result if external powered */
static u_char external_result[RESULT_MAX_LEN] = "0.00 V";

static u_char * battery(u_char *length)
{
    *length = RESULT_MAX_LEN;

    // check supply
    if (!btn_external_power())
    {
        int mvolt = btn_bat_measure(10);

        sprintf_P(result, PSTR("%d.%02d V"), mvolt / 1000, (mvolt % 1000)/10);

        return result;
    }
    else
        return external_result;
}

struct attr_metadata battery_metadata = \
    {NULL, battery, NULL, NULL, RESULT_MAX_LEN, ID, desc};
```

```
//
// BTnut: attr_heap.c
//

/**
* \date 27.07.2005
*
* \author Mustafa Yuecel <yuecelm@ee.ethz.ch>
*
* \brief attribute retrieving example with available heap
*/

#define ID 0x0002
#define DESC "Available Heap"
#define RESULT_MAX_LEN 17

#include <string.h>
#include <stdio.h>
#include <sys/heap.h>
#include "attr.h"
#include "attr_heap.h"

static u_char desc[strlen(DESC) + 1] = DESC;

static u_char result[RESULT_MAX_LEN];

static u_char * heap(u_char *length)
{
    sprintf_P(result, PSTR("%u bytes"), NutHeapAvailable());

    *length = (u_char) strlen(result) + 1;

    // fill rest with null bytes if you wish...

    return result;
}

struct attr_metadata heap_metadata = \
    {NULL, heap, NULL, NULL, RESULT_MAX_LEN, ID, desc};
```

Jaws Command Specification

November 28, 2005

This provides an overview on the available terminal commands. Important, changed or new commands are described in separate tables. Command output beginning with a colon is intended for being parsed by the JAWS graphical user interface.

1 Connection Manager Commands

```
cm id
cm relcons
cm inq
cm autinq [ 0|1 [0|1] ]
cm aiperiod [ SECONDS [0|1] ]
cm aitime [ SECONDS [0|1] ]
cm maxsl [1-7]
cm blink
cm disconall
```

2 Transport Manager Commands

```
tp openvc xx:xx:xx:xx:xx:xx
tp send <vc-handle> <len>
tp psend <vc-handle> <len> <#pkts> <pause 100ms>
tp clsnd xx:xx:xx:xx:xx:xx <len>
tp close <vc-handle>
tp vt (vc table)
tp ht (host table)
tp rc (reliable con)
tp bc
tp hd (host delete)
tp trace xx:xx:xx:xx:xx:xx
tp con xx:xx:xx:xx:xx:xx
tp rname xx:xx:xx:xx:xx:xx
tp cmd <ttl> xx:xx:xx:xx:xx:xx
tp verbose [1|0]
tp wd [reset]
```

con-state 0 = unconnected, 1 = neighbor is my master, 2 = neighbor is my slave, 3 = denied connection

Command	tp con <addr>
Output (async)	:T <source-addr> <num-entries> :TE <1st neighbor addr> <con-state> <NRSSI> :TE <2nd neighbor addr> <con-state> <NRSSI> ...

Table 1: Topology Request

NRSSI Negative RSSI in dB. Range: [0..100] (0=perfect, 100=bad)

Command	tp trace <addr>
Output (async)	:TR 1 <1st hop-addr> :TR 2 <2nd hop-addr> ... :TR N <addr>

Table 2: Trace: The target node sends a reply packet back to the host. On the way to the host each intermediate node appends its address to the packet.

Command	tp wd [reset]

Table 3: Transport watchdog: After 10 minutes without incoming packet the node is reset. This command prints or resets the watchdog statistics, i.e. how many times the watchdog has been triggered.

3 DSN Commands

```
dsn testpkt <vc-handle> <nr>
dsn time <vc-handle> <num> <len> <sleep>
dsn cmd <trans-id> [<addr>] [<cmd>]
dsn cdist on|off|run
dsn sendlog <addr> [<class> [<mask>]]
dsn logfilter <class> [<mask>]
```

trans-id Unique id used by GUI/user to map cmd result pkts to issued cmds.

seq-nr Sequence number always starting at zero.

total-len Length of the complete `<remote-cmd-output>` in bytes.

len Length of the following `<remote-cmd-output>` part in bytes.

4 Target Adapter Commands

```
tg cmd <cmd>
tg flash
```

Command	`dsn cdist run`

Table 4: Start code distribution immediately on local node.

Command	`dsn sendlog <host-addr> <log-class> <log-level>`
Output (on specified host node)	`:DL <source-addr> <log-class> <log-level> <length> <log-data>`

Table 5: Send stored log entries to a specified host address.

Command	`dsn logfilter <host-addr> <log-class> <log-level>`
Output (on specified host node)	`:DL <source-addr> <log-class> <log-level> <length> <log-data>`

Table 6: Permanently send new log entries to a host address according to log class and log levels specified. Deactivated by choosing log level zero. Omitting log class/level returns current setup.

Command	`dsn cmd <trans-id> [<host-addr>] [<remote-cmd>]`
Input	`<remote-cmd>`
Output (async)	`:C <source-addr> <trans-id> failed: <reason>` `:C <source-addr> <trans-id> completed: <total-len>`
Output (async)	`:CO <source-addr> <trans-id> <seq-nr> <len> <remote-cmd-output>`

Table 7: Remote command execution.

```
tg get bat
tg get fuses
tg get status
tg get version
tg reset
tg set fuses
tg set power on|off
```

Command	`tg flash`
Output 1	`:TF ok`
Output 2	`:TF failed: <reason>`

Table 8: Reprogram the target.

Command	`tg get fuses`
Output 1	`:TGF ok: 0xEEHHLL`
Output 2	`:TGF failed: <reason>`

Table 9: Reads extended, high and low fuse bytes.

Command	`tg set fuses`
Output 1	`:TSF ok: 0xEEHHLL`
Output 2	`:TSF failed: <reason>`

Table 10: Sets the standard fuses: `0xFF408E`.

5 Monitor Commands

```
mon bat
mon cmd <cmd>
mon heap
mon irq [<nr>]
mon net
mon reg <reg/port/mem-addr>
mon timers
mon threads
```

6 Logging Commands

```
log show [<class> [<mask]]
log clear
log logmask <class> [<mask>]
log verbosity <class> [<mask>]
```

Command	`tg get bat`
Output	`:TB <voltage> V`

Table 11: Print target battery voltage.

Command	`tg get version`
Output 1	`:TV YYYYMMDD-hhmm`
Output 2	`:TV unknown`

Table 12: Print target version. If no target monitor support is compiled into the target application, the string `unknown` is printed..

7 Other Commands

```
loadhex <ver> <type: 0=dsn, 1=tg> [<name>]
blink
get bat
reset
xbank get proginfo
xbank get status
xbank set progname <name>
xbank set progtype dsn|tg
xbank set progver <ver>
```

version Time encoded as 32-bit integer.

type Program type. 0 = dsn, 1 = target

name Program name.

voltage Format example: `2.56 V`

8 JAWS Commands

```
jaws get version
jaws get name
jaws set mode dsn|gui
jaws set name <name>
```

Command	`tg cmd <cmd>`

Table 13: Execute arbitrary terminal commands on the target.

Command	`tg reset`

Table 14: Reset the target.

Command	`tg set power on | off`

Table 15: Control target power. If target runs on batteries, this command should not be used.

Command	`log show <log-class> <log-level>`
Output	`<log buffer entries>`

Table 16: Output log buffer to terminal.

Command	`loadhex <version> [<type> <name>]`
Output	`ready to receive hex data, press enter for quit`
Input	program hex data
Output 1	`:LH completed: <nl> lines read`
Output 2	`:LH failed: <reason>`

Table 17: Load program code stored in Intel HEX file to local storage memory.

Command	`get bat`
Output 1	`:B <voltage> V`
Output 2	`:B external`

Table 18: Print battery voltage.

Command	`xbank get proginfo`
Output 1	`:XNP`

Output 2	`:XPT <progtype>`
	`:XPN <progname>`
	`:XPV <version>`
	`:XPS <progsize>`
	`:XPB <boot-addr>`

Table 19: Print information about program in storage memory.

Command	`jaws get version`
Output	`:JV YYMMDD-hhmm`

Table 20: Print JAWS program version.

Command	`jaws set mode dsn|gui`

Table 21: Set JAWS mode. Use this command to configure the local node as normal DSN node or as GUI node. Defines, whether the DSN adapter board or the USBprog adapter board can be attached.

Java-Sourcen

L2CAPCommunication

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: proj/yuecelm/L2CAPCommunication

RemoteProgrammer

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: proj/yuecelm/RemoteProgrammer

BTnodeInspector

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: proj/yuecelm/BTnodeInspector

TestCases (für L2CAPCommunication und RemoteProgrammer)

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: proj/yuecelm/TestCases

C-Sourcen

RemoteProgramming (Demo)

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: btnut/app/bt-remoteprog

AttributeRetrieving (Bibliotheken und Demo)

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: proj/yuecelm/attr-retrieving

Metadaten für OpenEmbedded

RemoteProgramming, AttributeRetrieving & TopologyDrawing

- CVS-Server: cvs.sourceforge.net
- CVS-Root: /cvsroot/btnode
- Modul: prof/yuecelm
- Revision-Tag: bbfiles