

**Michael Lerjen  
Christian Zbinden**

## **Reconfigurable Bluetooth Ethernet Bridge**



**Diplomarbeit DA-2002.01**

**WS 2001/2002**

**Betreuer:**

Jan Beutel  
Herbert Walder

**Professor:**

Prof. Dr. Lothar Thiele



# Vorwort

*“Harald Bluetooth was a viking and King of Denmark between 940 and 981. One of his skills was getting people to talk to each other, and during his rule Denmark and Norway were christianized and united. Today, Bluetooth wireless technology enables devices to talk to each other, but this time by means of a low-cost short-range radio link.”<sup>1</sup>*

*- Ericsson Mobile Communications -*

Während der Name Bluetooth bis ins zehnte Jahrhundert zurückreicht, gewinnt die gleichnamige Technologie Bluetooth zunehmend an Bedeutung.

So haben wir uns entschlossen unsere Diplomarbeit auf diesem Gebiet durchzuführen. Einerseits haben uns die Möglichkeiten von Bluetooth fasziniert, aber gewisse Limitationen auch herausgefordert, uns mit Bluetooth zu befassen.

Unsere Arbeit war sehr vielfältig. So haben wir uns sowohl mit Hardware als auch mit Software beschäftigt. Den Hardwareteil haben wir auf einem FPGA realisiert. Auf der Softwareseite haben wir uns mit hardwarenahen Linux-Kernel-Programmierung, aber auch mit Software im Benutzerbereich auseinandergesetzt. Zudem haben wir uns mit allen Schichten von Kommunikations-Protokollen beschäftigt.

Wir möchten vor allem unseren Betreuern Jan Beutel und Herbert Walder sowie Professor Lothar Thiele danken, welche diese interessante Arbeit erst ermöglicht haben.

Zürich, im März 2002,

Michael Lerjen und Christian Zbinden

---

<sup>1</sup>Im September 1999 stellte Ericsson in Lund (SE) eine Gedenktafel zur Erinnerung an Harald Bluetooth auf.



## DIPLOMARBEIT

für

Christian Zbinden und Michael Lerjen

Betreuer: Jan Beutel, Herbert Walder, Christian Plessl

Ausgabe: 17. September 2001

Abgabe: 25. Januar 2002

**Reconfigurable Bluetooth Ethernet Bridge****Einleitung**

Die elektronische Geräte um uns herum werden immer mehr und dringen in jeden Bereich unseres Lebens ein. Dabei wird die Kommunikation immer wichtiger um Funktionen zwischen Geräten auszutauschen und zu teilen. Der Trend geht dabei klar zu drahtlosen Kommunikationsschnittstellen. Herkömmliche Netzwerke bestehen meistens aus einer Basisstation und verschiedenen mobilen Geräten die dann sternförmig miteinander kommunizieren.

Bluetooth [12, 10] ist ein aktueller Kommunikationsstandard, der es ermöglicht auf kurzer Distanz Daten- und Audioverbindungen zwischen mobilen Geräten, PDA's, PC's, Telefonen, Druckern, Kopfhörern und vielen anderen Geräten aufzubauen. Solche Ad-hoc Netzwerke können dann aus mehreren Piconetzen mit bis zu 8 Geräten ein Scatternetz aufbauen. Weitere Designziele von Bluetooth sind eine kleine Baugrösse, geringe Implementationskosten, geringer Stromverbrauch sowie Sicherheit, Interoperabilität und Zuverlässigkeit im öffentlichen ISM Band auf 2.4 GHz.

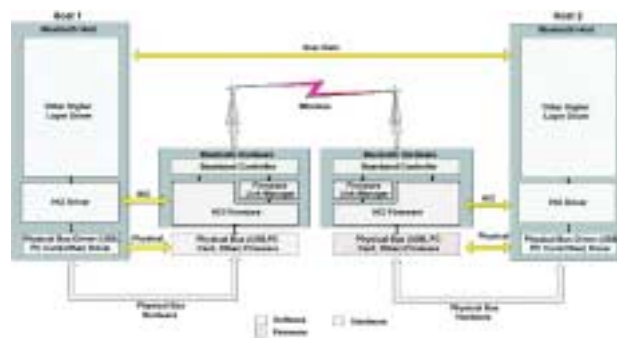


Abbildung 1: Bluetooth Protokollschichten und Partitionierung

Dabei ist Bluetooth so entworfen worden, das Geräte in funktionell in ein Frontend (z.B. Bluetooth Modul) und ein Backend (Host) partitioniert werden können (sieheq Abbildung ). Dabei ist das Frontend für die physikalische Verbindung und die damit verbundene Signalverarbeitung und das Backend für die logischen Verbindungen, die

Verbindungen zu Applikationen, Benutzern und Betriebssystemen in Anlehnung an das OSI Schichtenmodell zuständig. Front- und Backend kommunizieren dabei über das asynchrone Host Controller Interface (HCI) das für verschiedene Transport Schichten spezifiziert ist. Verschiedene Verbindungsmodi werden Profile [11, 9] genannt und ermöglichen heterogene Netzwerkstrukturen und Dienste.

Existierende Lösungen für Basisstationen oder Access Points gehen meist von einem Mikroprozessorsystem aus, das die gesamte Funktionalität eines Access Points bereit stellt. Dies kann z. B. ein Linux System sein, das einen Gerätetreiber für ein Bluetooth Frontend und eine Netzwerkanbindung unterhält [13, 6, 2, 15, 14].

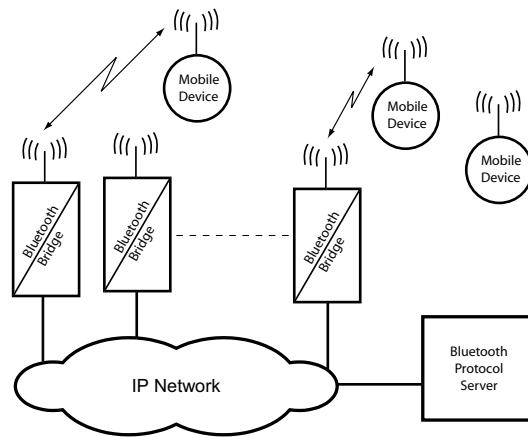


Abbildung 2: Verteilte Bluetooth Access Points mit zentralem Protokoll Server

In dieser Arbeit sollen Möglichkeiten zur Partitionierung und Verteilung von eines Bluetooth Access Point Frontends untersucht werden (siehe Abbildung ). Ziel ist es die verteilten Komponenten mehrer Access Points optimal auszulasten bzw. zu minimieren und ein zentrales Management zu ermöglichen. Um verschiedene Implementationsvarianten auch auszuprobieren soll als zentrale Komponente ein FPGA zum Einsatz kommen.

Hierzu soll das LAN Access Profile/RFCOMM auf einer FPGA Plattform implementiert werden die via Ethernet von einem Protocol Server unter Linux betrieben wird (siehe Abbildung ). Als Entwicklungsplattform steht ein XSV Virtex Prototyping Board zur Verfügung [7].



Abbildung 3: XSV Virtex Prototyping Board Übersicht

Ziel dieser Diplomarbeit ist es eine funktionierende Ethernet Bluetooth Bridge auf einem FPGA aufzubauen.

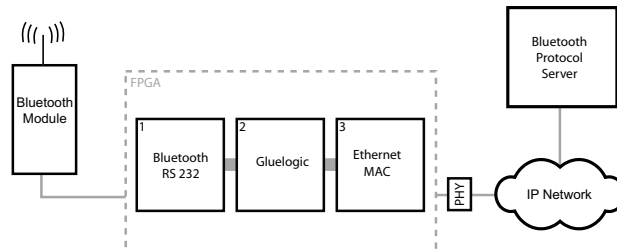


Abbildung 4: Funktionsübersicht der Bluetooth Bridge

## Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine sowohl zeitlich wie auch thematisch fest [19]. Erarbeiten Sie in Absprache mit dem Betreuer ein Pflichtenheft.
2. Machen Sie sich mit den am Institut und bei Prof. Mattern (Smart It's) bereits durchgeführten Arbeiten [15, 14, 16, 18, 8, 17, 4] vertraut. Es sollten möglichst viele Synergien aus schon durchgeführten Arbeiten genutzt werden.
3. Arbeiten Sie sich in die Softwareentwicklungsumgebung des BlueZ Bluetooth Protocol Stacks, Virtueller Tunnel und des Linux Kernels ein [13, 5, 2]. In der Bluetooth Protokoll Spezifikation sind dazu vor allem Part C, D, F.1 und H relevant [10]. Machen Sie sich mit den Mechanismen und Tools (Gnu Toolchain [3], Anjuta [1]) vertraut.
4. Arbeiten Sie sich in die FPGA Entwicklungsumgebung ein. Testen Sie die Entwicklungsplattform.
5. Machen Sie sich mit existierenden IP Bausteinen für FPGA's vertraut. Überlegen Sie welche Vor- und Nachteile der die verschiedene Angebotenen Lösungen bieten. Welche Funktionen z.B. eines Ethernetkontrollers werden Sie benötigen?
6. Zerlegen Sie die gesamte Funktionalität der Ethernet Bluetooth Bridge in Funktionsblöcke (Module) und definieren Sie die Schnittstellen, Datenflüsse (zwischen den Blöcken) und lokale Datenhaltung der einzelnen Module. Gehen Sie dabei hierarchisch vor (top-down) und berücksichtigen Sie folgende Aspekte:
  - Funktionale Anforderungen
  - Implementierbarkeit auf der Vorhandenen Infrastruktur (XESS-Board)
  - Wiederverwendbarkeit der Module
7. Dazu sollen Sie die folgenden Punkte bearbeiten:
  - Welche Funktionsblöcke/Module entstehen pro Hierarchiestufe?
  - Wie gestalten sich die Schnittstellen?
  - Welche Daten müssen wo gespeichert werden?
  - Welche Daten müssen an Folgeblöcke übergeben werden? Datenmenge?
  - Wie wird Flow-Control realisiert?
  - Wie wird die Datenintegrität gewährleistet (QoS IP, Packet loss, etc.). Welches Protololl wird verwendet? (TCP? Eigenes Protokoll?)
  - Welche Parameter beeinflussen den max. Durchsatz der EBB? Welches sind dabei die Schlüsselstellen im Design? Wo sind die Bottle-necks?
  - Welche Randbedingungen gelten (z.B. Vollständigkeit des IP-Stacks, Länge der ARP-Tabelle, etc.)
  - Wie wird die EBB konfiguriert? (MAC-Adresse, IP-Adresse, Baudrate, Flow-Control? statisch / dynamisch?)
  - Wo und wie können Debugging-Möglichkeiten eingesetzt werden?

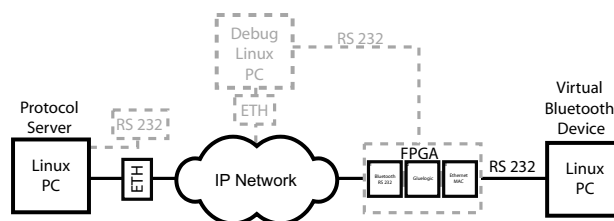
ommand Complete(0x0e)

8. Entwickeln Sie ein Konzept für Service Discovery und Handover von einem Access Point zum nächsten.
9. Schreiben Sie eine Kontrollapplikation sowie einen angepassten Treiber für den Bluetooth Protokoll Server. Versuchen Sie die in einer Semesterarbeit am TIK [17] entwickelte Multihop Funktionalität miteinfließen zu lassen.
10. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

### Phasen der Arbeit

Die im folgenden beschriebenen Phasen der Arbeit sollen als Richtlinie für den Entwicklungsprozess dienen. Sie können wahlweise auch ohne die FPGA Entwicklungsplattform realisiert werden indem man diese durch einen weiteren PC mit den erforderlichen Schnittstellen ersetzt.

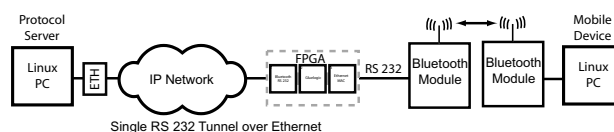
#### Einfacher Tunnel über RS-232



Hier soll in einem ersten Versuch eine Ethernet/RS-232 Verbindung aufgebaut und getestet werden. Beachten Sie das beim Tunneln einer RS-232 Verbindung keine Daten verloren gehen dürfen. Ermitteln Sie Performance und Robustheit einer solchen Verbindung in einem separaten Subnetz, und bei Routes über verschiedene Netzsegmente mit unterschiedlich viel Verkehr.

In der Konfiguration mit FPGA dient diese Konfiguration dazu den FPGA Entwurf zu testen. Weiterhin können Sie so verschiedene Konfigurationen, Protokolle und virtuelle Devices ausprobieren.

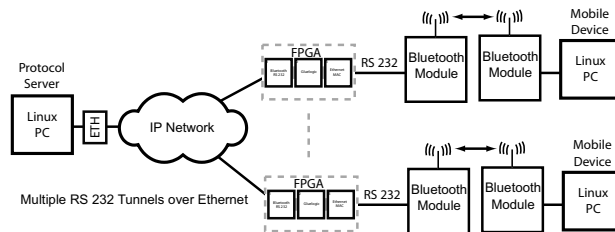
#### Einfacher Tunnel über Bluetooth



Testaufbau für einen einzelnen PPP Link über Bluetooth, wahlweise mit und ohne die FPGA Entwicklungsplattform.

Hier soll vorerst nur der HCI Transport Layer getunnelt werden. Versuchen Sie sowohl asymmetrische wie symmetrische Gerätekonfigurationen. Ermitteln Sie Performance und Robustheit des Bluetooth Protokolls, bzw. der HCI Schnittstelle. In dieser Konfiguration lassen sich Service Discovery sowie der Multihop DSR Protokoll Layer implementieren. Achten Sie dabei auf die folgende Erweiterung ihres Konzeptes mit mehreren Access Points.

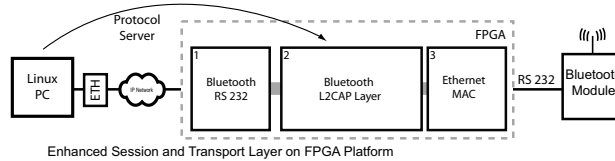
### Mehrfacher Tunnel über Bluetooth



Testaufbau für mehrer Access Points die von einem Protokoll Server aus bedient werden. Überprüfen Sie ob es notwendig ist für jeden Access Point einen eigenen Prozess/Treiber zu starten, oder ob sich dieses in der vorhandene Struktur multiplexen lässt.

Hier kann anschliessend Roaming/Handoff sowie Service Management ausprobiert werden.

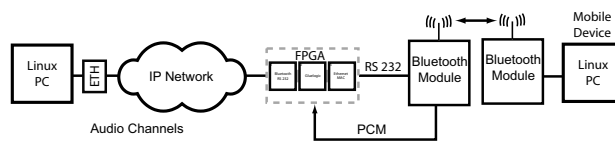
### Implementation des L2CAP Mechanismus im FPGA



Auslagerung des L2CAP Mechanismus (Session Layer) aus dem Protokoll Server in die verteilten Access Points. Ermitteln Sie ob es sinnvoll und möglich ist, dem FPGA-Access Point weitere Bluetooth Protokoll Funktionen zu übertragen.

Definieren Sie dazu eine neue Treiberschnittstelle und ein eigenes Tunnelprotokoll. Wägen Sie genau die Vor- und Nachteile bei der Systempartitionierung ab. Sie können sich dazu gut am BlueZ Treiber orientieren.

### Optional: Audioverbindung mit Bluetooth



Versuchen Sie eine synchrone Audio Verbindung mit Bluetooth zu erstellen und diese vom Audio Codec auf dem Entwicklungs Board wandeln zu lassen.

## Durchführung der Diplomarbeit

### Allgemeines

- Sie verfügen über PC's mit Linux/Windows für Softwareentwicklung und Test. Falls damit Probleme auftauchen wenden Sie sich an Ihren Betreuer.
- Stellen Sie Ihr Projekt zu Beginn der Semesterarbeit in einem Kurzvortrag vor und präsentieren Sie die erarbeiteten Resultate am Schluss im Rahmen des Institutskolloquiums Ende Semester.
- Besprechen Sie Ihr Vorgehen regelmässig mit Ihren Betreuern.

### Abgabe

- Geben Sie zwei unterschriebene Exemplare des Berichtes spätestens am 25. Januar 2002 dem betreuenden Assistenten oder seinen Stellvertreter ab. Diese Aufgabenstellung soll vorne im Bericht eingefügt werden.
- Räumen Sie Ihre Rechnerkonten soweit auf, dass nur noch die relevanten Source- und Objectfiles, Konfigurationsfiles, benötigten Directorystrukturen usw. bestehen bleiben. Der Programmcode sowie die Filestruktur soll ausreichen dokumentiert sein. Eine spätere Anschlussarbeit soll auf dem hinterlassenen Stand aufbauen können.

### Literatur

- [1] Anjuta IDE. <http://anjuta.sourceforge.net>.
- [2] Linux Kernel Archive. <http://www.kernel.org>.
- [3] The Gnu Project. <http://www.gnu.org>.
- [4] The Smart Its Project. <http://www.smart-its.org>.
- [5] VTUN - Virtual Tunnels over TCP/IP Networks. <http://vtun.sourceforge.net>.
- [6] Jan Beutel and Maksim Krasnyanskiy. *Linux BlueZ Howto*, 2001.
- [7] XESS Corporation. *XSV Board V1.0 Manual*.
- [8] Damon Fenacci, Marcel Grob, Peter Zberg, and Simon Künzli. Konzept Heim Netzwerk. Master's thesis, ETH Zürich, TIK, 2000.
- [9] Bluetooth Special Interest Group. *Bluetooth Assigned Numbers, v.1.1*, February 2001.
- [10] Bluetooth Special Interest Group. *Specification of the Bluetooth System - Core, v.1.1*, February 2001.
- [11] Bluetooth Special Interest Group. *Specification of the Bluetooth System - Profiles, v.1.1*, February 2001.
- [12] Jaap C. Haartsen. The Bluetooth Radio System. *IEEE Personal Communications*, February 2000.
- [13] Maksim Krasnyanskiy. BlueZ, Bluetooth protocol stack for Linux. <http://bluez.sourceforge.net>.
- [14] H. Mathys. Bluetooth Stack for the Palmpilot. Master's thesis, ETH Zürich, TIK, 2001.
- [15] H. Mathys, A. Pellmont, and A. Petralia. Home Networking - Implementation (Übertitel: DMA - Drop a Message Anywhere). Master's thesis, ETH Zürich, TIK, June 2000.
- [16] Thomas Moser and Lukas Karrer. The EventCollector Concept. Master's thesis, ETH Zürich, Distributed Systems Group, 2001.
- [17] Riccardo Semadeni and Lars Wernli. Bluetooth Unleashed, Wireless Netzwerke ohne Grenzen. Master's thesis, ETH Zürich, TIK, July 2001.

[18] Sascha Wulff and Marco Wirz. Bluetooth Networking - Protocol and Application Development. Master's thesis, ETH Zürich, TIK, February 2001.

[19] Eckart Zitzler. Studien- und Diplomarbeiten, Merkblatt für Studenten und Betreuer. ETH Zürich, TIK, March 1998.

Date	Section	Changes
Sept. 19, 2001		Initial version
Sept. 22, 2001		Refined version, to be discussed with Herbert and Christian
Oct. 2, 2001		New stylesheet
Oct. 22, 2001		Herbert's changes

Tabelle 1: Revision History



# Abstract

## **Introduction:**

Bluetooth is an emerging technology for short-range wireless links. It is designed to connect virtually any digital devices without proprietary cables. So it provides an universal bridge to existing networks, a peripheral interface, and a mechanism to build small wireless ad-hoc networks. A drawback is that all the communication is somehow limited to radio range, because the standard does not define any mechanism to forward messages over multiple Bluetooth devices.

This project is a further step towards the total integration of Bluetooth into data networks. With *Ethernet to Bluetooth Bridges* it is possible to equip whole buildings with Bluetooth infrastructure based on already existing IP-networks. To a roaming user, these *Ethernet to Bluetooth Bridges* can provide services like positioning, interconnectivity with the Internet and connection links that will be routed over multiple Bluetooth devices.

## **Aims and Goals:**

This project should provide a prototype of a *Bluetooth Ethernet Bridge* and demonstrate the feasibility of the mentioned concept.

To simplify the implementation of the mentioned services, the whole Bluetooth infrastructure should be managed by a central protocol server.

The *Ethernet to Bluetooth Bridge* should be implemented in hardware using a FPGA prototyping-board. The use of a FPGA allows not only to experiment with different implementations, but also to evaluate the use of reconfigurable hardware.

In a second part of our project, we analyzed the feasibility of task footprint transformations on Xilinx FPGAs. We used the JBits-SDK to manipulate the configuration bitstreams.

## **Results:**

A prototype of the *Bluetooth Ethernet Bridge* is working but limited to certain conditions.

A demonstration application, that measures the quality of all Bluetooth links of a *Ethernet to Bluetooth Bridge*, was implemented. This application builds not only the groundwork for services like positioning and access-points with handover but also shows the feasibility of the whole concept.

---

We also managed to demonstrate footprint transformation on a small design with a counter and a comparator task. However, it was not possible to transform larger tasks due to the limited routing capabilities of JBits.

**Further Work:**

The usability and the robustness of the prototype can be improved, e.g. the use of DHCP for the configuration of the *Bluetooth Ethernet Bridge* would relieve the user of any manual configuration.

Based on the demonstration application, all the mentioned services can be implemented. Especially the integration of the *Service Discovery Protocol* would improve the convenience of the system.

To further advance in the field of footprint transformations, there is a need for better tools. They must be provided by FPGA vendors as their development presumes very thorough knowledge of the internal structures.

# Inhalt

<i>1: Einleitung</i>	<i>1</i>
1.1 Vision . . . . .	1
1.2 Ethernet to Bluetooth Bridge . . . . .	2
1.2.1 Konzept . . . . .	2
1.2.2 FPGA-Implementation der E2B-Bridge . . . . .	2
1.2.3 Herausforderungen . . . . .	3
1.3 Strukturierung des Berichts . . . . .	3
<i>2: Background</i>	<i>5</i>
2.1 Bluetooth . . . . .	5
2.1.1 Überblick . . . . .	5
2.1.2 Protokoll Stack . . . . .	7
2.1.3 Verbindungs-Aufbau . . . . .	10
2.1.4 Limitationen . . . . .	11
2.2 Eingesetzte Bluetooth-Module . . . . .	12
2.3 Hardware Plattform . . . . .	12
2.3.1 XESS Prototyp-Board . . . . .	12
2.3.2 Virtex XCV800 FPGA . . . . .	14
2.3.3 Entwicklungs-Software . . . . .	14
<i>3: Konzept - MC-Protokoll</i>	<i>15</i>
3.1 Partitionierung: PC ↔ FPGA . . . . .	15
3.1.1 Aufteilung des Bluetooth-Stacks . . . . .	15
3.1.2 Aufteilung der Kommunikationsintelligenz . . . . .	17
3.2 Grundgedanken zum Tunneln von RS-232 über Netzwerke . . . . .	18
3.3 MC-Protokoll . . . . .	19
3.3.1 Einordnung in bestehende Protokolle . . . . .	19
3.3.2 Grundelemente . . . . .	19
3.3.3 Verbindungs-Zustandsmaschine . . . . .	20
3.3.4 Header - Felder . . . . .	21
3.3.5 Flusskontrolle und Management der Windowgrößen . . . . .	24
3.3.6 Fehlermanagement . . . . .	26

3.3.7	Implementationsdetails . . . . .	26
3.4	Erweiterbarkeit des MC-Protokolls . . . . .	27
4:	<i>Implementation Software</i> . . . . .	29
4.1	Spezifikationen . . . . .	29
4.2	Vorhandene Software - <i>Bluez</i> . . . . .	29
4.2.1	Allgemeines Funktionsprinzip . . . . .	30
4.2.2	Schnittstelle zur Applikation: Socket Interface . . . . .	31
4.2.3	Schnittstelle zum Host Controller Transport Layer . . . . .	32
4.3	Integrationskonzept der neuen Software . . . . .	33
4.3.1	Konzept im Kernel-Bereich . . . . .	33
4.3.2	Konzept im User-Bereich . . . . .	33
4.4	Konzept des Softwarepakets <i>Netty</i> . . . . .	34
4.4.1	Netzwerk-Modul . . . . .	35
4.4.2	Prozessfluss von <i>Netty</i> . . . . .	36
4.4.3	Datenstrukturen . . . . .	37
4.4.4	Implementationsdetails . . . . .	37
5:	<i>Implementation Hardware</i> . . . . .	41
5.1	Designspace . . . . .	41
5.2	Architektur . . . . .	43
5.3	Der IP-Stack . . . . .	43
5.3.1	Auswahl eines Designs . . . . .	43
5.3.2	Probleme und Fehler . . . . .	45
5.3.3	Verbesserungen des Designs . . . . .	46
5.3.4	Erweiterung um das MC-Protokoll . . . . .	47
5.4	RS-232/Glue-Logic . . . . .	50
5.4.1	RS-232 . . . . .	50
5.4.2	Glue-Logic . . . . .	51
5.5	Simulation . . . . .	53
6:	<i>Resultate und Fazit</i> . . . . .	55
6.1	Test-Szenarien . . . . .	55
6.1.1	Test des MC-Protokolls . . . . .	55
6.1.2	Demonstrations-Applikation . . . . .	56
6.2	Erfahrungen und Einschränkungen . . . . .	56
6.2.1	E2B-Bridge . . . . .	56
6.2.2	Netty . . . . .	57
6.2.3	Bluetooth-Module . . . . .	57
6.2.4	Bluez . . . . .	58
6.3	Fazit . . . . .	58

<i>7: Dynamische Rekonfiguration des FPGAs</i>	59
7.1 Die Xilinx-Virtex Architektur und Rekonfiguration . . . . .	59
7.2 JBits . . . . .	61
7.2.1 Das JBits-Paket . . . . .	61
7.2.2 Wie wird JBits angewendet? . . . . .	61
7.3 Anwendung: Multitasking in einem FPGA . . . . .	62
7.4 Task-Transformationen . . . . .	63
7.4.1 Relokation . . . . .	63
7.4.2 Footprint-Transformationen . . . . .	63
7.5 Versuche mit JBits . . . . .	65
7.5.1 Zusammensetzen zweier Designs . . . . .	65
7.5.2 Verbinden und Verschieben zweier Tasks . . . . .	67
7.5.3 Footprint-Transformationen . . . . .	68
7.6 Resultate / Fazit . . . . .	69
<i>8: Ausblick und Dank</i>	71
8.1 Ausblick . . . . .	71
8.1.1 Software . . . . .	71
8.1.2 E2B-Bridge . . . . .	71
8.1.3 Footprint-Transformationen . . . . .	72
8.2 Dank . . . . .	72

## Anhang

<i>A: Howto</i>	75
A.1 Struktur der beiliegenden CD . . . . .	75
A.2 Beschreibung der ausführbaren Programme . . . . .	75
A.2.1 Bluez - Programme . . . . .	76
A.2.2 Programmpaket Netty . . . . .	79
A.3 Inbetriebnahme . . . . .	81
A.3.1 Bluez . . . . .	81
A.3.2 Netty . . . . .	81
A.3.3 E2B-Bridge . . . . .	81
A.3.4 Rekonfiguration . . . . .	82
<i>B: Dokumentation der Software Netty</i>	83
B.1 File - Struktur . . . . .	83
B.2 Konfigurations-Möglichkeiten im File <i>netty.h</i> . . . . .	84
B.3 Compilieren von Netty . . . . .	84

<i>C: Zustands-Diagramme des IP-Stacks</i>	<i>87</i>
C.1 Ethernet-Sender (ethernetsnd.vhd) . . . . .	87
C.2 Ethernet-Empfänger (ethernet.vhd) . . . . .	88
C.3 ARP-Sender (arpsnd.vhd) . . . . .	88
C.4 ARP-Empfänger (arp3.vhd) . . . . .	88
C.5 Internet-Sender (internetsnd.vhd) . . . . .	89
C.6 Internet-Empfänger (internet.vhd) . . . . .	89
C.7 ICMP (icmp.vhd) . . . . .	90
<i>D: Bluetooth</i>	<i>91</i>
D.1 Grafische Darstellung des gesamten Bluetooth-Stacks . . . . .	91
D.2 Traces von HCI - Sequenzen . . . . .	94
D.2.1 Init Sequence . . . . .	94
D.2.2 RSSI . . . . .	96
D.2.3 Change BT-Address . . . . .	97
D.2.4 Inquiry . . . . .	98
D.2.5 hci_config . . . . .	98
D.2.6 l2ping . . . . .	100
D.2.7 l2test . . . . .	102
<i>E: Vortrag</i>	<i>107</i>
<i>Literaturverzeichnis</i>	<i>113</i>

# Tabellenverzeichnis

2-1	Eckdaten des Bluetooth Radio-Layers . . . . .	8
2-2	Vergleich zwischen den verschiedenen eingesetzten BT-Modulen . . . .	12
3-1	Zustandsabfolge des MC-Protokolls (Sicht des Host Computers) . . . . .	22
3-2	Zustandsabfolge des MC-Protokolls (Sicht der E2B-Bridge) . . . . .	23
3-3	Eckdaten zum Design des Sliding Windows . . . . .	25
4-1	Mögliche Ereignisse in der Software <i>Netty</i> . . . . .	36
5-1	Eigenschaften des IP-Stacks der Universität von Queensland . . . . .	44
B-1	Konfigurations-Möglichkeiten im File <i>netty.h</i> . . . . .	85



# Abbildungsverzeichnis

1-1	Grobkonzept zur Verwirklichung der Bluetooth Vision . . . . .	2
2-1	Netzwerktopologien bei Bluetooth . . . . .	6
2-2	Bluetooth Protokoll-Stack . . . . .	7
2-3	Aufteilung in BT-Modul und BT-Host . . . . .	8
2-4	Zeitlicher Verlauf von SCO- und ACL-Funkverbindungen . . . . .	9
2-5	Blockdiagramm des XESS XSV800 Boards . . . . .	13
3-1	Partitionierung des Bluetooth-Stacks . . . . .	16
3-2	Standard-Partitionierung des Bluetooth-Stacks . . . . .	16
3-3	Partitionierungsvariante mit Schnittstelle HCI . . . . .	17
3-4	Partitionierungsvariante mit L2CAP auf dem FPGA . . . . .	17
3-5	Einordnung des MCP in bestehende Protokolle . . . . .	19
3-6	Prinzip des 3-Way-Handshake . . . . .	20
3-7	Zustandsdiagramm von TCP . . . . .	20
3-8	Zustandsdiagramm des MCP . . . . .	21
3-9	Headerfelder des MCP . . . . .	21
4-1	Struktur des Bluez-Stacks . . . . .	30
4-2	Struktur gemäss dem Kernel-Bereich Konzept . . . . .	34
4-3	Integrationskonzept von <i>Netty</i> . . . . .	35
4-4	Prozessfluss von <i>Netty</i> . . . . .	36
4-5	Screenshots vom GUI . . . . .	39
5-1	Blockdiagramm des Designs . . . . .	43
5-2	Blockdiagramm des MCP-Empfängers . . . . .	47
5-3	Blockdiagramm des MCP-Senders . . . . .	48
5-4	Zustands-Diagramm der MCP Zustandsmaschine . . . . .	49
5-5	Interface des IP-Stacks mit MCP . . . . .	49
5-6	Übertragung eines Bytes mit dem RS232-Protokoll . . . . .	50
5-7	Das Interface des RS232-Blocks . . . . .	50
5-8	Timing-Diagramm des RS232-Empfängers . . . . .	51
5-9	Timing-Diagramm des RS232-Senders . . . . .	51

5-10	Blockdiagramm der Glue-Logik . . . . .	52
5-11	Datenfluss-Diagramm des Testbenchs . . . . .	53
6-1	Demonstrations-Applikation . . . . .	56
7-1	Schematische Darstellung des Aufbaus eines Virtex FPGAs . . . . .	60
7-2	Prozessor-Core mit Audio-Codec . . . . .	63
7-3	Unterteilung des IP-Stacks in Subtasks . . . . .	64
7-4	Beispiel für eine CLB-Transformation . . . . .	65
7-5	Funktion des Programms Test1.java . . . . .	66
7-6	Störende Routing-Reste . . . . .	66
7-7	Funktion des Programms Test2.java . . . . .	67
7-8	Funktion des Programms Test3.java . . . . .	68
A-1	File-Struktur der beigelegten CD . . . . .	75
B-1	File-Struktur von Netty . . . . .	83

# 1

## *Einleitung*

In diesem Kapitel wird ein erster Überblick über unsere Arbeit gegeben. Dies geschieht in den folgenden drei Schritten. In Kapitel 1.1 werden die Idee und die Motivation unserer Arbeit erläutert, während in Kapitel 1.2 das Konzept erklärt wird, wie die Idee umgesetzt werden kann. In Kapitel 1.3 wird auf die Strukturierung der weiteren Teile des Berichts eingegangen.

### *1.1 Vision*

Die Idee hinter unserer Arbeit ist, dass Bluetooth vollständig ins alltägliche Leben integriert werden soll. Eine im ganzen Gebäude fest installierte Bluetooth-Infrastruktur soll es möglich machen, dass mit Bluetooth ganz neue Dienste angeboten werden können. So wird das Ersetzen von lästigen Kabeln zwischen einem mobilen Computer und dessen Peripherie nicht mehr die Hauptaufgabe von Bluetooth bleiben, sondern es können auch die folgenden Dienste angeboten werden:

- **Positionsbestimmung (und darauf aufbauende Dienste):**

Wenn die Qualität aller möglichen Verbindungen zwischen einem mobilen Gerät und der fest installierten Bluetooth-Infrastruktur gemessen wird, kann die Position des mobilen Geräts berechnet werden. Die berechnete Position kann nun sowohl vom System zur Überwachung benutzt werden als auch dem mobilen Benutzer zur Orientierungshilfe mitgeteilt werden. Auf der Positionsbestimmung aufbauend können auch ortsabhängige Dienste angeboten werden.

Die Positionsbestimmung mit Bluetooth ist dabei umso wertvoller, da GPS innerhalb von Gebäuden nicht zufriedenstellend eingesetzt werden kann.

- **Roaming mit Handover:**

Die fest installierte Bluetooth-Infrastruktur bietet Access-Points (AP) zu anderen Datennetzen an. Dabei ist ein Roaming mit Handover möglich. Wenn also ein mobiles Bluetooth-Gerät via einen AP eine FTP-Verbindung ins Internet aufbaut und sich dann weiterbewegt, so soll die Verbindung transparent

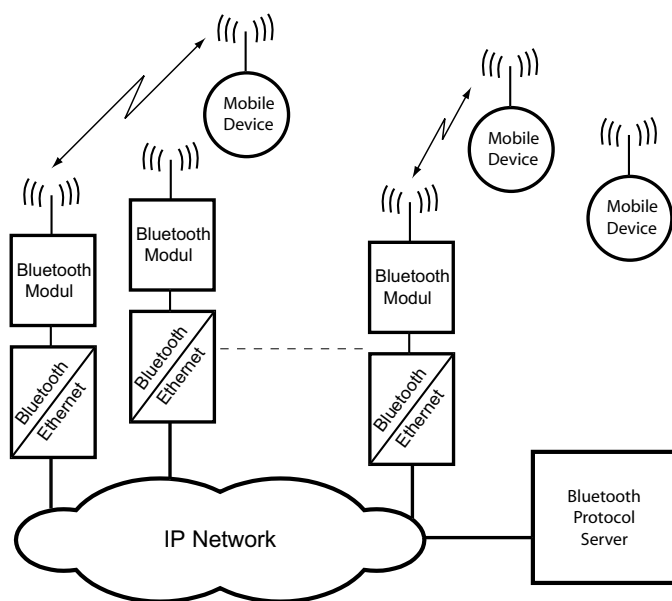
zu einem besser erreichbaren AP weitergereicht werden können, ohne dass der FTP-Server etwas davon bemerkt.

- **Weiterleiten von Paketen (Multihop):**

Das in [7] vorgestellte Konzept zum Weiterleiten von Paketen kann erweitert werden. So können die Pakete nun auch über die drahtgebundene Infrastruktur weitergeleitet werden und somit Daten auch über grössere Distanzen ausgetauscht werden.

## 1.2 Ethernet to Bluetooth Bridge

### 1.2.1 Konzept



*Abbildung 1-1  
Verbindungsablauf  
gemäss Grobkonzept:  
Ein mobiles Gerät  
nimmt Verbindung auf  
mit einem  
Bluetooth-Modul,  
welches fest installiert  
und an eine  
E2B-Bridge  
angeschlossen ist. Die  
ankommenden Daten  
werden dabei von der  
E2B-Bridge über ein  
IP-Netzwerk zu einem  
Protokoll-Server  
weitergeleitet, wo sie  
weiter verarbeitet  
werden.*

Das Grobkonzept, wie diese Bluetooth-Vision verwirklicht werden soll, ist in Abbildung 1-1 dargestellt.

Um eine möglichst einfache und flexible Verkabelung der Bluetooth-Infrastruktur zu ermöglichen, soll diese mit einem IP-Netzwerk, welches in vielen Gebäuden bereits vorhanden ist, verbunden werden. Das Management der Infrastruktur soll von einem zentralen Protokoll-Server übernommen werden, was die Implementation der gewünschten Dienste erst möglich macht oder wenigstens stark vereinfacht.

Als Bindeglied zwischen dem IP-Netzwerk und dem Bluetooth-Modul wird eine *Ethernet to Bluetooth Bridge (E2BB)* benötigt, dessen Entwicklung der Hauptbestandteil unserer Arbeit darstellt.

### 1.2.2 FPGA-Implementation der E2B-Bridge

Existierende Lösungen von Access-Points gehen meist von einem Mikroprozessor-System aus, welches die gesamte Funktionalität des Access-Points implementiert.

In unserer Arbeit soll aber die E2B-Bridge in Hardware realisiert werden. Dabei wird ein FPGA-Prototyping-Board verwendet. Dies hat den Vorteil, dass mehrere Implementationen getestet werden können.

In einem weiteren Schritt konnten mit diesem FPGA-Prototyping-Board auch Aspekte des *Reconfigurable Computing* untersucht werden. Dabei ging es insbesondere um die Machbarkeit von Footprint-Transformationen auf einem Virtex FPGA. Dies stellte den zweiten Hauptbestandteil unserer Arbeit dar.

#### 1.2.3 Herausforderungen

Eine Herausforderung unserer Arbeit ist es, eine optimale Partitionierung des Bluetooth-Stacks zu finden. Dabei wird die Frage untersucht, welche Funktionen sich besser auf dem FPGA implementieren lassen und welche besser auf dem Protokoll-Server.

Des weiteren muss die Kommunikation zwischen der E2B-Bridge und dem Server mit einem geeigneten Kommunikations-Protokoll sichergestellt werden.

### 1.3 Strukturierung des Berichts

Die weiteren Teile des Bericht sind folgendermassen gegliedert:

- Kapitel 2 beschreibt die Grundlagen und die Hilfsmittel, die uns für diese Arbeit zur Verfügung standen.
- Kapitel 3 zeigt auf, wie wir den Protokoll-Stack partitioniert haben und wie das von uns verwendete Protokoll aufgebaut ist.
- Die Implementation wird in den Kapiteln 4 (Software) und 5 (Hardware) beschrieben.
- Die Resultate haben wir in Kapitel 6 zusammengefasst.
- Kapitel 7 ist dem zweiten Gebiet, dass wir in unserer Arbeit untersucht haben, gewidmet: Den Footprint-Transformationen auf dem FPGA.
- Kapitel 8 schliesslich bietet einen Ausblick auf mögliche Erweiterungen und Fortsetzungen unserer Arbeit.



# 2

## *Background*

In diesem Kapitel werden die Hintergrund-Informationen vermittelt, die für das Verständnis unserer Arbeit notwendig sind. So behandelt Kapitel 2.1 die allgemeine Idee hinter Bluetooth, während Kapitel 2.2 konkret auf die verfügbare Bluetooth-Hardware eingeht. Im Kapitel 2.3 wird dann die FPGA-Entwicklungs-Plattform beschrieben.

### *2.1 Bluetooth*

#### *2.1.1 Überblick*

##### *2.1.1.1 Geschichtliche Entwicklung*

Im Jahre 1998 wurde die *Bluetooth Special Interest Group (SIG)* von Ericsson, IBM, Intel, Nokia und Toshiba mit dem Ziel gegründet, eine offene Spezifikation für eine kabellose Kommunikationslösung für kleine Reichweiten zu entwickeln. In den folgenden Jahren haben sich über 1900 Firmen der SIG angeschlossen (unter anderem 3COM, Microsoft, Lucent und Motorola).

##### *2.1.1.2 Motivation*

Der Grundgedanke, der zur Entwicklung von Bluetooth führte, ist die Idee des *ubiquitous computing*.

All die peripheren Geräte sollen kabellos mit möglichst kleinem Aufwand und möglichst ohne Konfiguration von Seiten des Benutzers miteinander kommunizieren können. Der Standard soll global akzeptiert werden, so dass die Kommunikation überall und mit allen Geräten, unabhängig von der Marke, stattfinden kann.

##### *2.1.1.3 Grundgedanke*

Bluetooth ist ein Standard für Funkübertragungen auf kurze Distanzen. Dabei bietet die Technologie auch einen kabellosen Zugang zu LANs, PSTN, mobiler Telefonie und zum Internet. Beim Standard wird vor allem Wert gelegt auf Robustheit, Einfachheit, wenig Stromverbrauch und geringe Kosten.

Bluetooth Geräte kommunizieren über Kurzdistanz-AdHoc-Netzwerke, Piconetze genannt. Das Piconetz wird charakterisiert durch einen Master, welcher den Zugriff auf das Netz regelt, und bis zu 7 Slave-Geräten. Dabei kann ein Gerät gleichzeitig Mitglied mehrerer Piconetze sein (Scatternetz). Diese Netzwerke werden dynamisch und automatisch aufgebaut, wenn Bluetooth-Geräte in die Funkreichweite gelangen.

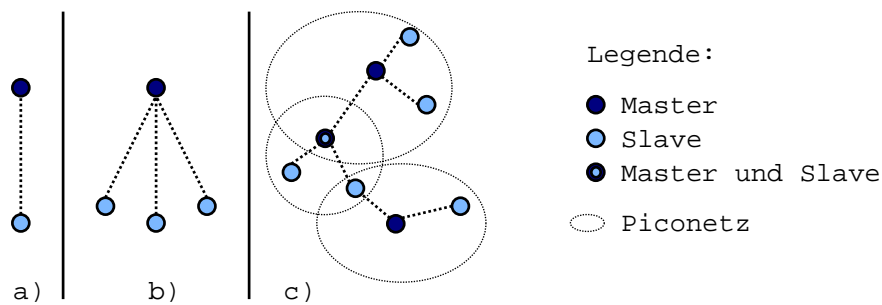


Abbildung 2-1

Netzwerktopologien bei Bluetooth:

a) Point-to-Point: nur ein Slave b) Piconetz: mehrere Slaves c) Scatternetz: Mehrere Piconetze, die sich überlappen.

#### 2.1.1.4 Technische Herausforderungen

##### - Frequenzband:

Damit der Standard universal eingesetzt werden kann, muss die Frequenz in einem unlicenzierten Band liegen. Deshalb wurde das Industrial, Scientific and Medical (ISM) Band gewählt. Die Herausforderung besteht darin, das System robust gegen Interferenzen von anderen Geräten zu machen. Dabei sind nicht nur andere Benutzer des Frequenzbands zu beachten, sondern z.B. auch Mikrowellenöfen. Im besten Fall sendet jedes Gerät mit der minimal notwendigen Leistung, um das Rauschen für andere Geräte nicht zu erhöhen.

##### - Bewegte Geräte:

Der Sender muss sich an schnell verändernde Umgebungen anpassen können. Dabei müssen sowohl Mehrwegwellenausbreitung (multipath fading) als auch sich verändernde Netzwerktopologien behandelt werden.

##### - Beschränkte Ressourcen:

Der Aufwand zur Implementation des Standards soll klein sein, um eine einfache Integration in Handhelds und mobile Geräte zu gewährleisten. Dabei sind vor allem die beschränkten Ressourcen wie Rechenleistung und Speicherplatz zu beachten.

##### - Beschränkte Energiereserve:

Bluetooth soll im Vergleich zum Gerät, in das die Bluetooth-Funktionalität integriert werden soll, nur wenig Energie benötigen.

##### - Verbindungsmanagement:

Da die Anzahl und Art der Geräte, die sich in Funkreichweite befinden, ziem-

lich schnell ändert, ist es umständlich, wenn die Verbindungen vom Benutzer verwaltet werden müssen.

- Sicherheit:

Da Bluetooth-Geräte sensible Daten enthalten und kommunizieren, müssen diese Daten geschützt werden.

## 2.1.2 Protokoll Stack

Das Design von Bluetooth ist in mehrere fast unabhängige Schichten aufgeteilt. (siehe auch [1], [2] und [8])

### 2.1.2.1 Überblick

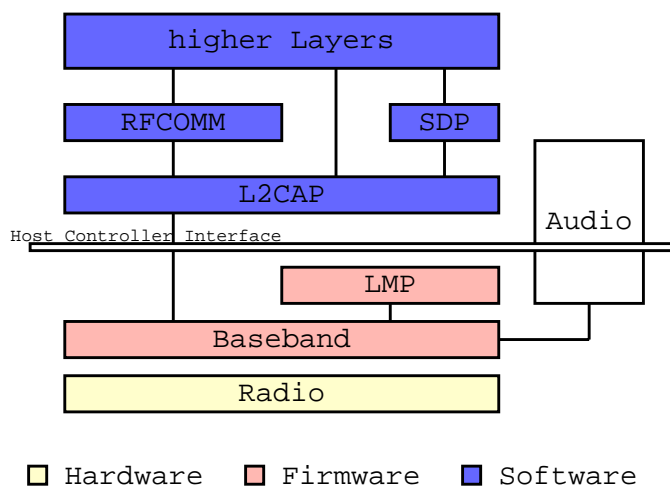


Abbildung 2-2  
Bluetooth  
Protokoll-Stack

Der Bluetooth Protokoll-Stack wird in Abbildung 2-2 grafisch dargestellt. Dieser besteht aus einem *Radio-Layer*, welcher sich zuunterst befindet, und die physikalische Verbindung zu einem andern Modul übernimmt. Darüber befindet sich der *Baseband-Layer* und der *Link Manager Protocol (LMP) Layer*, welche für den Aufbau und Verwaltung von Verbindungen zu anderen Modulen verantwortlich sind. Diese drei Schichten werden normalerweise in Hardware/Firmware realisiert (siehe Abbildung 2-3).

Der *Host Controller Layer* ist nötig als Schnittstelle zwischen der Bluetooth Hardware und dem *Logical Link Control and Adaptation Protocol (L2CAP)*. Diese Zwischenschicht ist nur nötig, falls der L2CAP-Layer in Software auf dem Host realisiert wird. Falls der L2CAP-Layer aber auf dem Bluetooth-Modul realisiert wird, ist dieser Zwischenlayer nicht nötig, da der L2CAP-Layer direkt mit dem LMP-Layer kommunizieren kann. Der Application-Layer befindet sich als höchster Layer über dem L2CAP-Layer.

### 2.1.2.2 Radio-Layer

Der Radio-Layer operiert im 2.4 GHz Band. In fast allen Ländern liegt der genaue Frequenzbereich, den Bluetooth benutzt, im Bereich von 2400 - 2483.5 MHz. Das verwendete Bandspreizverfahren ist *Frequency Hopping (FH)*. Um Interferenzen

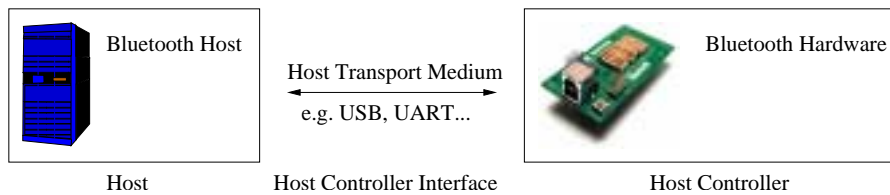


Abbildung 2-3

Normalerweise wird der Bluetooth Protokoll-Stack in zwei Teile aufgeteilt:

Bluetooth Hardware mit den unteren Protokollschichten und Host Controller mit den oberen komplexeren Protokollschichten. Die Kommunikation zwischen den beiden Teilen geschieht mittels der Host Controller Interface (HCI)-Schicht

mit anderen Netzen zu vermeiden, wurden kurze Paketlängen und schnelles Hopping gewählt. Die nominale Hopraterate liegt bei 1600 Hops pro Sekunde, was einer Slotlänge von  $625 \mu\text{sec}$  entspricht. Bezüglich der Sendeleistung der Bluetoothgeräte gibt es zwei Kategorien. Geräte mit 10 m Reichweite und 0dBm Sendeleistung und solche mit 100 m Reichweite und 20dBm Sendeleistung.

Bandspreiztechnik	Frequency Hopping
Frequenzen	$2402+k \text{ MHz}$ (mit $k = 0,1,\dots,78$ )
Hopping Rate	1600 Hops/sec

Tabelle 2-1: Die wichtigsten technischen Daten des Bluetooth Radio-Layers.

### 2.1.2.3 Baseband

Der Baseband-Layer kontrolliert den Radio-Layer. So werden die Frequenzhopping-Sequenzen in diesem Layer bestimmt. Ebenso stellt der Baseband-Layer die Funkfunktionalität zur Verfügung, die gebraucht wird, um die Clocks der Geräte zu synchronisieren und Funkverbindungen aufzubauen. Es können prinzipiell zwei Arten von Funkverbindungen aufgebaut werden:

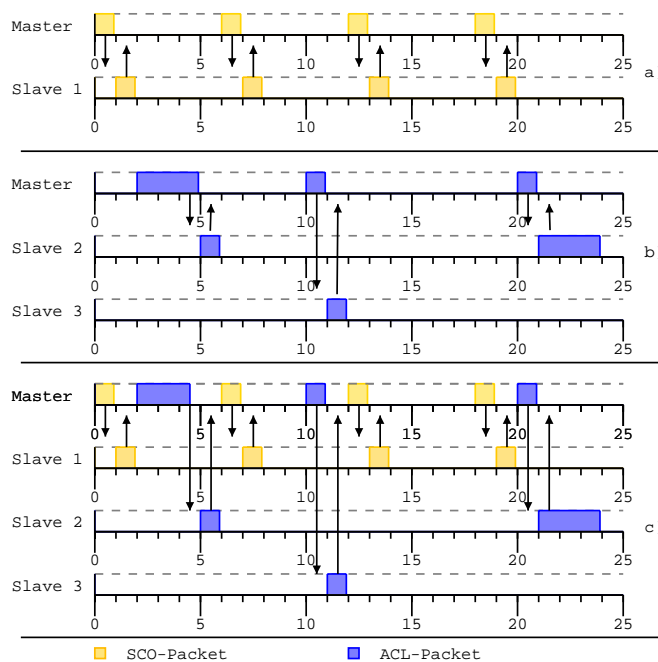
- *Synchronous Connection Oriented (SCO)*:  
Diese Verbindungsart wird für synchrone Datenübertragung verwendet, wobei diese Verbindung typischerweise für Audioübertragungen verwendet wird.
- *Asynchronous Connection Less (ACL)*:  
Diese Verbindungsart wird für Datenübertragungen verwendet, die keine synchrone Verbindung benötigen.

Ein weiterer Bestandteil des Baseband-Layers sind Inquiries, um die Bluetooth-Adressen der sich in Funkreichweite befindenden Geräte zu suchen.

### 2.1.2.4 Link Manager Protocol (LMP)

Die drei Hauptaufgaben des LMP sind Piconetz-Management, Konfiguration der Funkverbindungen und Sicherheitsfunktionen.

Ein Piconetz ist eine Gruppe von Geräten, die mit einem gemeinsamen Kommunikationskanal verbunden ist, der durch eine eindeutige Hopping-Sequenz charakterisiert wird. Ein Gerät (normalerweise das Gerät, das die Verbindung initiiert) ist der Master, welcher die Hopping-Sequenz vorgibt. Bis zu sieben Slaves können mit dem Master verbunden sein. Der Zugriff innerhalb eines Piconetz wird vom Master folgendermassen verwaltet: Jeder Slave darf nur senden, wenn er im vorhergehenden Slot ein Paket vom Master empfangen hat. Dieser Ablauf wird in Abbildung 2-4 illustriert.



*Abbildung 2-4  
Zeitlicher Verlauf von  
SCO- und ACL-  
Funkverbindungen:  
a) SCO-Verbindung,  
b) ACL-Verbindungen,  
c) SCO und  
ACL-Verbindungen:  
Prinzipiell können die  
Pakete zwischen 1 und  
5 Slots lang sein. Für  
SCO-Verbindungen  
können Slots mit einer  
fixen Intervallzeit  
reserviert werden. Die  
übriggebliebenen Slots  
können dann für  
ACL-Verbindungen  
verwendet werden.*

Zwei oder mehr Geräte, die miteinander kommunizieren wollen, müssen selbständig ein Piconetz aufbauen. Dabei können Geräte Teil mehrerer Piconetze sein.

Die Aufgabe des LMP ist es, Slaves in das Piconetz zu integrieren und wieder zu entfernen. Ausserdem kann er die Rolle des Slaves und des Masters vertauschen, Funkverbindungen auf- und abbauen und den Zustand des Bluetooth-Moduls verwalten (low power mode, sniff, park ...).

### 2.1.2.5 Host Controller Interface (HCI)

Der HCI-Layer ist nur notwendig, falls der Protokoll-Stack in die zwei Teile Bluetooth-Hardware und Bluetooth-Host aufgeteilt wird (siehe auch Abbildung 2-3). Dann werden die Daten, die zum LMP- und Baseband-Layer geschickt werden sollen, über ein HCI-Transportmedium (wie z.B. USB) geschickt. Für diese

Übertragung ist auf beiden Seiten ein Treiber und ein einheitliches Datenformat nötig, welches vom HCI-Layer zur Verfügung gestellt wird.

#### *2.1.2.6 Logical Link Control and Adaptation Protocol (L2CAP)*

Die wesentlichen Aufgaben des L2CAP-Layers sind:

- Multiplexing:  
Das Protokoll muss mehreren Applikationen erlauben, eine Funkverbindung gleichzeitig zu verwenden.
- Segmentierung und Reassembly:  
Das Protokoll muss die Grösse der Pakete, die es von einem höheren Layer in Empfang nimmt, reduzieren auf die Grösse, die vom Baseband-Layer akzeptiert wird. L2CAP akzeptiert Paketgrößen bis 64kb, Baseband-Pakete können aber höchstens 2745 Bit gross sein. Für empfangene Pakete muss die umgekehrte Prozedur durchlaufen werden, das heisst, die Baseband-Pakete müssen wieder zusammengesetzt werden.
- Quality of Service (QoS):  
L2CAP erlaubt höheren Layern verschiedene Verbindungsparameter zu fordern. L2CAP überprüft, ob die Verbindung fähig ist, diese Parameter zu liefern.

Im Prinzip stellt L2CAP die Funktionen der Netzwerkschicht zur Verfügung.

#### *2.1.2.7 Service Discovery Protocol (SDP)*

Der SDP-Layer stellt Funktionen zur Verfügung, mit welchen eine Applikation erfahren kann, welche Services erreichbar sind und welcher speziellen Art die Services sind. Dabei ist er optimiert für die hochdynamische Bluetooth-Umgebung.

#### *2.1.2.8 Höhere Layer*

Bluetooth sieht mehrere Möglichkeiten vor, bestehende Protokolle an den Bluetooth-Stack anzubinden, unter anderem die folgenden:

- serielle Protokolle:  
Mit dem *RFCOMM-Layer* stellt Bluetooth eine Emulation von seriellen Ports zur Verfügung. Mit diesem Layer sind z.B. PPP-Verbindungen und darauf aufbauend TCP-Verbindungen möglich
- Telefonie:  
Mit der *Telephony Specification (TCS)* besteht eine Möglichkeit zur Anbindung von Telefonie-Applikationen.

### *2.1.3 Verbindungs-Aufbau*

Dieses Kapitel beschreibt die Schritte, die ausgeführt werden müssen, wenn zwei oder mehrere Bluetooth-Geräte eine Verbindung untereinander aufbauen wollen.

In unserem Beispiel wird von einem User ausgegangen, der eine E-Mail mit einem Bluetooth-fähigen Gerät verschicken will.

1. Inquiry:  
Das Gerät sucht nach anderen Geräten, die sich in Funkreichweite befinden und wählt eines davon aus.
2. Paging:  
Die beiden Kommunikationspartner synchronisieren sich aufeinander in Beziehung auf Clockoffset und Phase in der Hopping-Sequenz. Ebenso werden andere Initialisierungen durchgeführt.
3. Verbindungsaufbau:  
Der LMP-Layer baut eine Verbindung auf. Weil die Anwendung das Verschicken einer E-Mail ist, wird eine ACL-Verbindung verwendet.
4. Service Discovery:  
Der LMP-Layer verwendet das SDP, um herauszufinden, ob die Gegenstelle das Versenden von E-Mails und den Zugriff auf den entsprechenden Host unterstützt. In unserem Fall ist die Gegenstelle ein Access-Point, der dies unterstützt.
5. L2CAP-Verbindung:  
Mit den Informationen des SDP baut das Gerät nun eine L2CAP-Verbindung zum Access-Point auf. Diese Verbindung kann jetzt direkt von der Applikation benutzt werden, oder andere Protokolle können darauf aufgesetzt werden, wie z.B. RFCOMM.
6. RFCOMM:  
Eine RFCOMM-Verbindung wird aufgebaut. Diese Verbindung erlaubt es, Applikationen, die für eine serielle Schnittstelle geschrieben wurden, ohne Anpassungen zu benutzen.
7. PPP, TCP:  
Aufbauend auf der Emulation einer seriellen Schnittstelle können nun die selben Anwendungen zum Verschicken der E-Mail benutzt werden, welche für ein Modem verwendet werden.

#### 2.1.4 Limitationen

Die grösste konzeptionelle Einschränkung von Bluetooth ist, dass das Arbeitsumfeld eines Bluetooth-Gerät stets auf 10-100m beschränkt ist. Das Weiterleiten von Nachrichten ist nicht vorgesehen.

Ein gravierendes Problem ist auch, dass der Standard zwar schon seit langem besteht, funktionierende Bluetooth-Geräte und Software-Stacks aber noch kaum vorhanden sind.

Ebenso unterscheiden sich die verschiedenen Umsetzungen des Standards stark je nach Hersteller. Die angestrebte Kompatibilität zwischen den Geräten der verschiedenen Hersteller ist nicht vollkommen vorhanden.

Diese Diskrepanz zwischen der Bluetooth-Theorie in Form der Bluetooth-Spezifikation [1] und der Praxis in Form von käuflicher Hardware zeigt sich stark bei den von uns verwendeten Bluetooth-Modulen (siehe Kapitel 2.2).

## 2.2 Eingesetzte Bluetooth-Module

Alle von uns eingesetzten Bluetooth-Module sind von Ericsson. Dabei können die folgenden drei Typen unterschieden werden:

- ROK 101 007 (alte Module)  
Mit diesen Modulen, welche die Bluetooth-Spezifikation 1.0b erfüllen, sind nur Punkt-zu-Punkt-Verbindungen möglich.
- ROK 101 007 (neue Module)  
Mit diesen Modulen ist es möglich ein Piconetz mit einem Master und mehreren Slaves aufzubauen.
- ROK 101 008  
Mit diesen Modulen sind ebenfalls nur Punkt-zu-Punkt-Verbindungen möglich.

Die Eigenschaften, die wir während unserer Arbeit gefunden haben, sind in Tabelle 2.2 zusammengefasst.

<i>ROK 101 007 (alte Module)</i>	<i>ROK 101 007 (neue Module)</i>	<i>ROK 101 008</i>
Bluetooth-Standard 1.0b	Bluetooth-Standard 1.0b	Bluetooth-Standard 1.1
Punkt-zu-Punkt	Punkt-zu-Multipunkt (Piconetz)	Punkt-zu-Punkt
-	erhältlich auf Sigma Comtec Kit	erhältlich auf Sigma Comtec Kit
Audio (PCM & USB) (nicht getestet - laut Spezifikation)	Audio (PCM & USB) (nicht getestet - laut Spezifikation)	Support für 2 Audio-kanäle (nur PCM) (nicht getestet - laut Spezifikation)
USB	USB	kein USB Support
RSSI-Kommando implementiert	RSSI nicht implementiert (in neuerer Firmware behoben)	RSSI-Kommando implementiert
Klasse 2 Bluetooth Gerät (0dBm)		
max 460 kb/s über UART-Schnittstelle		

Tabelle 2-2: Vergleich zwischen den verschiedenen eingesetzten Bluetooth-Modulen

## 2.3 Hardware Plattform

### 2.3.1 XESS Prototyp-Board

Als Plattform für die Hardware-Entwicklung stand uns ein XSV800-Board von der Firma XESS zur Verfügung. Es enthält als Herzstück einen Xilinx Virtex XCV800

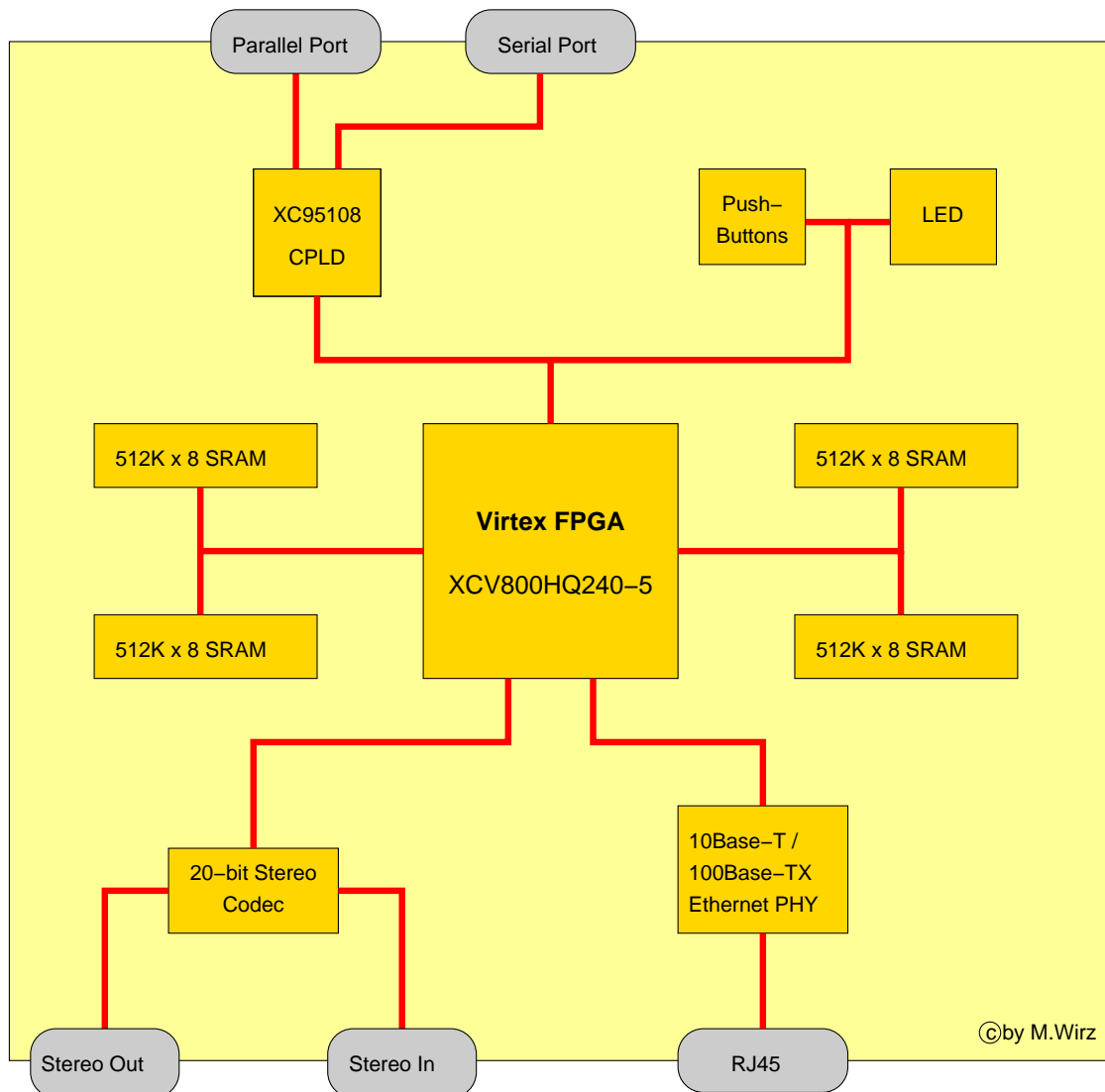


Abbildung 2-5  
Blockdiagramm des XESS XSV800 Boards

FPGA. Daneben besitzt es viele verschiedene Schnittstellen zur Aussenwelt:

- Parallel-Port
- Serieller Port (RS232)
- XChecker-Port mit JTAG-Interface
- USB 1.1-Port mit dazugehörigem PHY-Chip
- PS/2-Port
- 10/100 Mbps Ethernet Interface mit dazugehörigem PHY-Chip
- VGA-Anschluss mit einem 110MHz-Ramdac
- S-Video Eingang mit zugehörigem Decoder

- Audio In/Out Interface mit 20Bit A/D-Wandler mit bis zu 100kHz Samplerate

Daneben gibt es auf dem Board selber auch noch weitere Hardware:

- Zwei RAM-Bänke mit je 512K×8 Bit.
- 16 MBit Flash zum Speichern von Konfigurationsdaten
- Zwei Siebensegment-Anzeigen und eine Balkenanzeige mit 10 LEDs
- Vier Druckknöpfe und acht DIP-Schalter
- Einen programmierbaren Oszillator für die Taktgeneration <sup>1</sup>

Die Abbildung 2-5 zeigt ein Blockdiagramm mit den von uns benutzten Komponenten. Auf der Homepage von XESS [11] gibt es ausserdem ein Tutorial sowie einen reichhaltigen Fundus an Beispielprojekten, die frei heruntergeladen werden können.

### *2.3.2 Virtex XCV800 FPGA*

Der auf dem XESS-Board eingesetzte *Xilinx Virtex XCV800* hat eine Kapazität von 888K Gates und gehört damit zu den grösseren FPGAs der Virtex-Familie. Er ist in ein HQ-240-Gehäuse eingebaut. Einige Pins werden für die Spannungsversorgung und die Konfiguration benötigt, so dass noch 166 Pins für I/O zur Verfügung stehen.

Im Innern befinden sich total 4704 CLBs. Sie enthalten 18'816 Register und ebenso viele Lookup-Tables mit je vier Eingängen. Weitere Angaben können dem Datenblatt von Xilinx [12] entnommen werden.

### *2.3.3 Entwicklungs-Software*

#### *2.3.3.1 Synthese*

Zu Beginn unserer Arbeit benutzten wir Version 3.1i der *Xilinx Foundation Software* unter Windows. Danach haben wir auf Version 4.1i gewechselt, da diese schneller arbeitet. Der Wechsel klappte ohne Probleme. Auch die neue Version lief stabil. Ein Problem konnten wir jedoch mit dem *Floorplanner* feststellen. Beim Festlegen von Area-Constraints gab er eine zu kleine minimale Grösse an. Dadurch liessen sich Constraints festlegen, die gar nicht erreicht werden konnten, was nachher beim Place- und Route-Vorgang zu einem Abbruch mit Fehlermeldung führte. Ansonsten gab es keine weiteren Probleme.

#### *2.3.3.2 Simulation*

Zur funktionalen Simulation des VHDL-Codes benutzten wir das *Modelsim-Paket* von Mentor Graphics in der Version 5.5d. Es ist sehr mächtig in der Funktionalität und trotzdem sehr einfach und intuitiv zu bedienen. Wir haben es sowohl unter Windows als auch unter Solaris benutzt, wobei die Solaris-Version deutlich stabiler lief.

---

<sup>1</sup>Dieser scheint jedoch bei vielen Boards nicht zu funktionieren. XESS gibt zwar auf der Homepage Tipps für einen Workaround, bei unserem Board half jedoch nur das Anschliessen eines externen Oszillators.

# 3

## *Konzept* - MC-Protokoll

In diesem Kapitel wird genauer auf die Kommunikation zwischen der E2B-Bridge und dem Protokoll-Server eingegangen.

Das Konzept dieser Kommunikation wird in drei wesentlichen Schritten entworfen. Zuerst wird in Kapitel 3.1 bestimmt, welche Funktionalität im FPGA der E2B-Bridge und welche Funktionalität im Protokoll-Server implementiert werden soll. In einem zweiten Schritt wird in Kapitel 3.2 auf die Probleme eingegangen, die sich bei einer Übertragung von Bluetooth-Daten über ein IP-Netz grundsätzlich stellen werden. In Kapitel 3.3 wird dann das schlussendlich verwendete Kommunikations-Protokoll dargelegt.

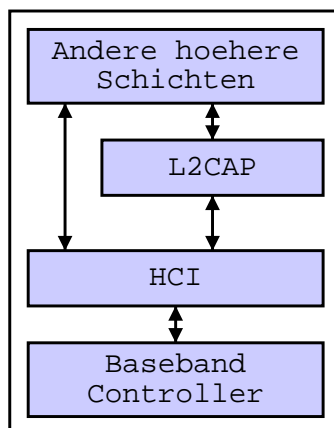
### *3.1 Partitionierung: PC ↔ FPGA*

Bei der Partitionierung müssen zwei Hauptpunkte entschieden werden. Erstens wie der Bluetooth-Stack auf die beiden Geräte Host-Computer und E2B-Bridge aufgeteilt wird. Die zweite Frage ist, wie die Kommunikation zwischen den beiden Teilsystemen ablaufen soll. Liegt die ganze Intelligenz auf der einen Seite und das andere Gerät muss nur ganz wenig Funktionalität aufweisen, oder ist es von Vorteil, wenn die Kommunikations-Intelligenz gleichmässig auf die beiden Kommunikationspartner verteilt ist?

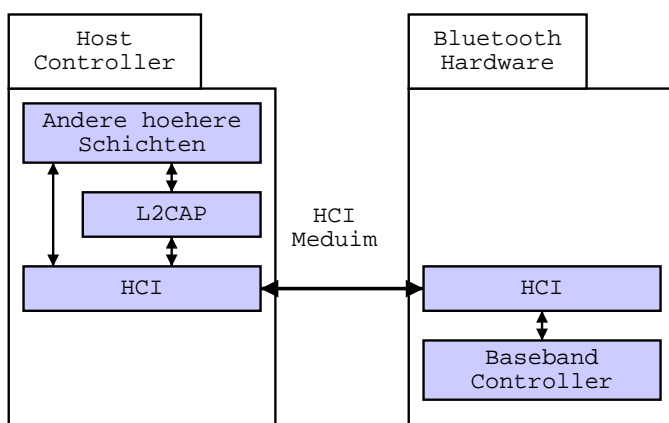
#### *3.1.1 Aufteilung des Bluetooth-Stacks*

Im Bluetooth-Standard ist die HCI-Schicht als Schnittstelle zwischen Bluetooth-Hardware und Host-Computer vorgesehen, aber eine Auslagerung weiterer Funktionalität auf die E2B-Bridge würde die Netzwerkbelastung verringern (siehe Abbildungen 3-1 und 3-2).

Dabei ist vor allem zu beachten, wie der Bluetooth-Stack auf die zusätzlichen Verzögerungen reagiert, die durch die Netzwerkverbindung verursacht werden. Erwartungsgemäss ist die Toleranz umso grösser, je höher im Protokoll-Stack die Trennung zwischen Host-Rechner und E2B-Bridge vorgenommen wird.



*Abbildung 3-1  
Partitionierungs-  
möglichkeiten des  
Bluetooth-Stacks:  
Diese Abbildung zeigt  
die wichtigsten  
Bluetooth-Schichten.  
Untersucht wird, wie  
diese am besten auf  
Host-Rechner und  
E2B-Bridge aufgeteilt  
werden können.*



*Abbildung 3-2  
Standard-  
Partitionierung des  
Bluetooth-Stacks:  
Diese Abbildung zeigt  
die standardmässige  
Aufteilung der  
Bluetooth-Schichten  
falls keine E2B-Bridge  
vorhanden ist.*

Da gewisse Applikationen direkt auf die HCI-Schicht zugreifen, muss auch unsere Aufteilung des Protokoll-Stacks dies zulassen. Das heisst, dass unsere Software eine Schnittstelle für HCI-Pakete bieten muss (siehe Abbildung 3-3). Das Kommunikations-Protokoll zwischen Host-Computer und E2B-Bridge muss dann in der Lage sein muss, HCI-Pakete zu übertragen.

Es ist aber denkbar, dass die L2CAP-Schicht ebenfalls auf die E2B-Bridge verlagert wird. Dann muss unsere Software eine zusätzliche Schnittstelle für L2CAP-Daten haben und das Kommunikations-Protokoll muss zusätzlich in der Lage sein, L2CAP-Kommandos und L2CAP-Daten zu übertragen (siehe Abbildung 3-4). Diese Variante mit der L2CAP-Schicht auf der E2B-Bridge bringt folgende Vor- und Nachteile:

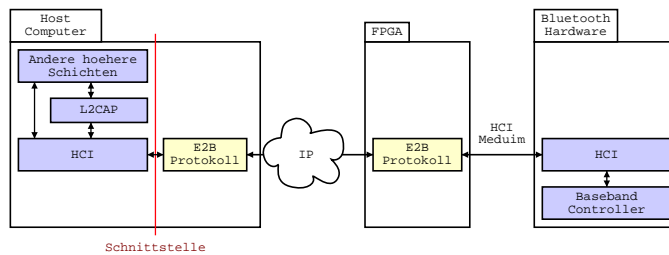
- Vorteile:

Durch die grössere Intelligenz in der E2B-Bridge müssen weniger Daten über das Netzwerk geschickt werden. Somit wird die Gesamtverzögerung verkleinert und zusätzlich nimmt die Netzwerkbelastung ab. Der ACL-Verbindungsaufbau kann eventuell von der E2B-Bridge selbständig abgewickelt werden, so dass die Gefahr weniger gross ist, auf Grund von Netzwerkverzögerungen ein Timeout auszulösen.

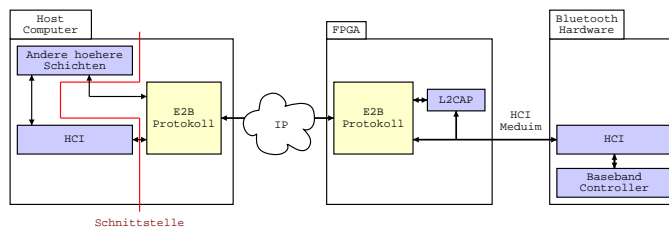
- Nachteile:

Die Implementation von L2CAP-Teilen im FPGA ist komplex, weil der Ablauf stark abhängig von den zu verarbeitenden Daten ist [6]. Denn je nach ACL-Kommando muss anders vorgegangen werden. Ebenso müssen die verschiedenen Programmteile synchronisiert und dadurch dynamisch Daten gepuffert werden. Das Kommunikations-Protokoll wird erheblich komplizierter, da sowohl HCI-Pakete als auch ACL-Pakete getrennt voneinander übermittelt werden müssen. Der Bluetooth-Stack auf Seite des Host-Computers muss stark angepasst werden.

Bei genauerem Überlegen verringert die Auslagerung der L2CAP-Schicht den Netzwerkverkehr nur unbedeutend, da die meisten Pakete doch bis zum Benutzer weitergereicht werden müssen. Dies sind sicher einmal alle Daten, aber ebenso ist für die meisten Verbindungen auch eine Interaktion mit einer Applikation nötig, so zum Beispiel betreffend der Parameter der Verbindung. Auf Grund dieser Argumente haben wir uns entschieden, die Partitionierung auf der HCI-Schicht vorzunehmen (siehe Abbildung 3-3), das Kommunikations-Protokoll aber so zu entwerfen, dass eine spätere Auslagerung der L2CAP-Schicht auf die E2B-Bridge (siehe Abbildung 3-4) trotzdem möglich bleibt.



*Abbildung 3-3  
Partitionierungsvariante  
mit Schnittstelle HCI:  
Der erste  
Implementations-  
Schritt trennt den  
Bluetooth-Stack an  
der Stelle des Host  
Controller Interfaces.*



*Abbildung 3-4  
Partitionierungsvariante  
mit L2CAP auf dem  
FPGA: Spätere Aus-  
bauoption, die mit  
unserem  
Kommunikations-  
Protokoll prinzipiell  
möglich ist.*

### 3.1.2 Aufteilung der Kommunikationsintelligenz

Da die Implementation der Kommunikationsintelligenz auf einem vollständigen Computer mit einem Betriebssystem und einer höheren Programmiersprache wie C++ einfacher zu gestalten ist als in Hardware auf einem FPGA, soll die Funktionalität soweit wie möglich auf der Seite des Computers implementiert werden. Aus Performancegründen soll aber von einer totalen Master-Slave-Beziehung mit Fehlerkontrolle und Flusskontrolle nur auf Seiten des Computers abgesehen werden.

## 3.2 Grundgedanken zum Tunneln von RS-232-Verbindungen über Netzwerke

Da die Trennung zwischen Host-Computer und E2B-Bridge auf der HCI-Schicht vollzogen wird, besteht die Grundaufgabe des Kommunikations-Protokolls darin, die seriellen Daten der HCI-Schicht über ein Netzwerk zu tunneln und auf der anderen Seite der nächsten Schicht des Bluetooth-Stacks weiterzugeben. Als Netzwerk soll ein IP-Netzwerk zum Einsatz kommen, da IP-Netzwerke weit verbreitet sind und sich damit dank den Routingmöglichkeiten auch grössere Distanzen überbrücken lassen. Die einzige Distanzlimitation wird in unserem Fall durch die maximale Netzwerkverzögerung gegeben. Da sich die Kommunikation über die serielle Schnittstelle grundsätzlich von der Kommunikation über ein Netzwerk unterscheidet, muss beim Design des Kommunikations-Protokolls einiges beachtet werden.

- Verzögerungszeiten:

Wie bereits erwähnt müssen die zusätzlichen Verzögerungen beim Design des Kommunikations-Protokolls berücksichtigt werden. So ist darauf zu achten, dass durch geeignete Massnahmen die Verzögerung durch das Netzwerk möglichst klein gehalten wird.

- Fehlertoleranz:

Da bei Übertragungen über ein serielles Kabel davon ausgegangen wird, dass keine Fehler auftreten, muss das E2B-Protokoll in geeigneter Weise eine Fehlerdetektion und Fehlerkorrektur vornehmen, da in einem Netzwerk davon ausgegangen werden muss, dass einzelne Pakete verloren gehen oder nicht fehlerfrei übertragen werden.

- Bytestream vs. Pakete:

Ein weiterer Unterschied ist, dass die Daten bei der seriellen Kommunikation byteweise in einem Stream übertragen werden, in Netzwerken aber Pakete verschickt werden. Dabei stellt sich die Frage nach der optimalen Grösse der Pakete, was eine Abwägung zwischen Overhead und Netzwerkverzögerung ist.

- geordnete vs. ungeordnete Daten:

Weil bei der seriellen Übertragungen alle Bytes auf dem gleichen Weg übertragen werden und keine Fehler auftreten können, werden die Daten auf der Empfangsseite in der gleichen Reihenfolge empfangen wie sie gesendet wurden. Dies ist in IP-Netzwerken nicht gewährleistet.

Da diese Probleme grundsätzlich bei jeder Netzwerkverbindung auftreten, gibt es auch schon einige Standardlösungen. So stellt TCP/IP [34] eine fehlerfreie Bytestream-Verbindung zwischen zwei Netzwerkendpunkten zur Verfügung. Da aber TCP/IP dazu konzipiert wurde, in verschiedenen Netzwerktopologien und Situationen einen guten Durchsatz zu erbringen, ist das Protokoll höchst dynamisch, was eine Implementation auf dem FPGA sehr aufwändig macht. Da viele Bestandteile von TCP/IP für unsere Kommunikation gar nicht zwingend notwendig sind, haben wir uns entschieden, ein eigenes einfacheres Protokoll (MC-Protokoll genannt)

zu entwickeln. Dieses ist im Gegensatz zum Streaming-Protokoll TCP/IP ein Paket-Protokoll, was die Implementierung auf einem FPGA stark vereinfacht.

### 3.3 MC-Protokoll

#### 3.3.1 Einordnung in bestehende Protokolle

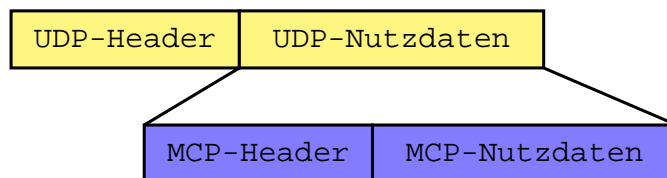


Abbildung 3-5  
MC-Protokoll

Unser Protokoll setzt auf UDP [33] auf. Die UDP-Nutzdaten bestehen einerseits aus dem MC-Header und den MC-Nutzdaten (siehe Abbildung 3-5), wobei die ersten acht Bytes der UDP-Nutzdaten für den MC-Header reserviert sind (für Headerdetails siehe Kapitel 3.3.4). Die MC-Nutzdaten bestehen immer aus einer abgeschlossenen Anzahl HCI-Pakete. Somit wird sichergestellt, dass allfällige Erweiterungen auf Seite der E2B-Bridge einfach vorgenommen werden können. Falls die E2B-Bridge eigene HCI-Pakete generiert, kann sie diese einfach zwischen den Daten des MC-Protokolls eingefügen, ohne die Daten des MC-Protokolls auf allfällige HCI-Paketgrenzen untersuchen zu müssen.

#### 3.3.2 Grundelemente

Ein notwendiges Element, um eine fehlertolerante Verbindung aufzubauen, sind einerseits *Sequenznummern*. Diese garantieren, dass die richtige Reihenfolge wieder hergestellt werden kann, auch wenn die Datenpakete in der falschen Ordnung eintreffen. Ebenso können Duplikate erkannt werden. Um richtig empfangene Pakete zu bestätigen, braucht es auch *Acknowledgenummern*, die zum Sender zurückgeschickt werden. Am Anfang einer Übertragung wissen die beiden Kommunikationspartner nicht, welche Sequenznummer von der Gegenstelle gerade gebraucht wird. Trotzdem muss aber sichergestellt werden, dass sowohl bereits das erste Datenpaket richtig empfangen wird, als auch Pakete einer anderen oder einer alten Verbindung ignoriert werden. Deshalb muss eine *Verbindungsaufbauprozedur* durchlaufen werden, während der sich die beiden Kommunikationspartner auf die zu verwendenden Sequenznummern einigen.

Diese drei Punkte müssen auf alle Fälle erfüllt werden, damit eine fehlertolerante Verbindung überhaupt möglich ist. Somit sind sie auch die Grundelemente des MC-Protokolls. Der nächste Schritt in unseren Überlegungen ist nun, welche weiteren Elemente von TCP/IP, die mit geringem Aufwand implementiert werden können und einen Mehrnutzen bringen, zusätzlich in unserem Protokoll implementiert werden sollen.

Eines davon ist die Implementierung eines *Sliding Window*. Wobei aber in unserem Protokoll nicht der freie Platz in Bytes zurückgemeldet wird wie es in TCP

geschieht, sondern die Anzahl der Pakete, die noch empfangen werden können. Verzichtet haben wir angesichts der aufwändigen Implementierbarkeit auf jegliche Art von *Überlastungsüberwachung* und *dynamischen Timeouts*.

### 3.3.3 Verbindungs-Zustandsmaschine

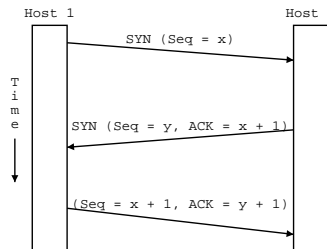


Abbildung 3-6  
Prinzip des 3-Way-Handshake, welches verhindert, dass veraltete Pakete den Verbindungsaufbau negativ beeinflussen können.

Das Verbindungsmanagement haben wir weitgehend von TCP/IP übernommen. So findet der Verbindungsaufbau ebenfalls über ein 3-Way-Handshake statt (siehe Abbildung 3-6). Auch die restlichen Zustandsabfolgen haben wir weitgehend übernommen. (Abbildung 3-8 zeigt unser Zustandsdiagramm, während Abbildung 3-7 zum Vergleich die Zustandsabfolgen von TCP zeigt).

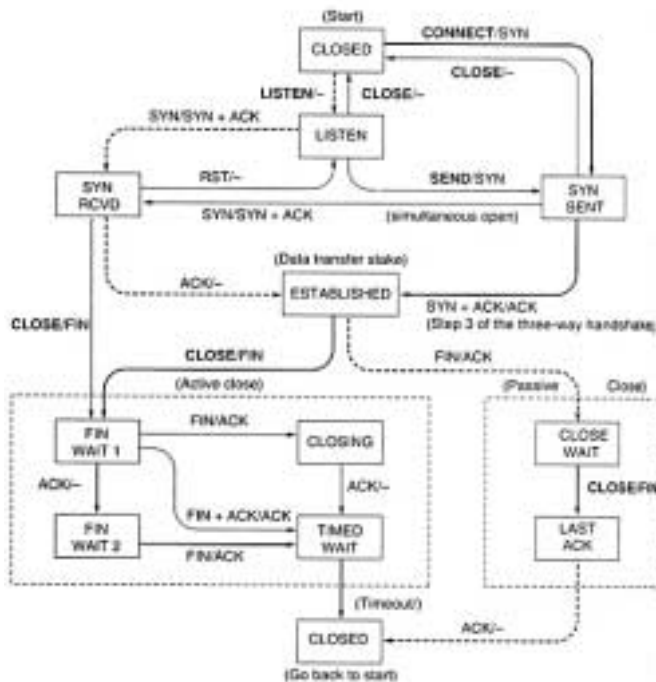


Abbildung 3-7  
Zustandsdiagramm von TCP

Die Anpassungen, die wir vorgenommen haben, beziehen sich vor allem auf die spezielle Situation, die wir für unsere Anwendung vorgefunden haben. So müssen Spezialfälle, wie z.B. wenn beide Kommunikationspartner versuchen, eine Verbindung aufzubauen, nicht berücksichtigt werden, denn die E2B-Bridge ist immer die Serverseite und der Host-Computer immer der Client, der die Verbindung aufbauen



Tabelle 3-1: Zustandsabfolge des MC-Protokolls (Sicht des Host Computers)

Host Computer:		
Zustand → Transition	Bedingungen	Aktion
CLOSED → SYN_SENT	-	Ein Paket mit gesetztem SYN-Bit wird zur E2B-Bridge geschickt. Bei der Sequenznummer (x) ist darauf zu achten, dass das Paket nicht mit alten noch vorhandenen Paketen verwechselt werden kann. Dies kann erreicht werden, indem mindestens so lange gewartet wird, bis die Timeouts aller noch vorhandenen Pakete abgelaufen sind, oder indem mit der nächst höheren als der im letzten Paket der alten Verbindung benutzten Sequenznummer fortgefahren wird.
SYN_SENT → ESTABLISHED	Ein Paket mit gesetzten SYN und ACK -Bits und der ACK-Nummer (x + 1) wird empfangen.	Beginn der Datenübertragung. Das erste Datenpaket hat die Sequenznummer (x + 1). Die ACK-Nummer ist die empfangene Sequenznummer der Gegenstelle inkrementiert um eins (y + 1).
ESTABLISHED → FIN_WAIT1	Die Verbindung wird vom Benutzer geschlossen (CLOSE-Befehl).	Der Kommunikationspartner wird mit einem Paket mit gesetztem FIN-Bit und mit der nächsten SEQ-Nummer (z) informiert. Dabei ist der Spezialfall zu beachten, dass kein Platz mehr im Sende-Window vorhanden ist. Dann muss mit dem Senden gewartet werden, bis wieder Platz vorhanden ist.
FIN_WAIT1 → TIME_WAIT	Ein Paket mit FIN-Bit, der SEQ-Nummer (u) und der ACK-Nummer, die das eigene gesendete FIN-Paket bestätigt (z + 1) wird empfangen.	Der IN- und OUT- Stack wird geleert (EMERGENCY CLOSE: der Verlust noch ausstehender Pakete wird akzeptiert, um den Abbau zu beschleunigen). Ein ACK-Paket wird zurückgeschickt. (seq = z + 1, ack = u + 1). Danach wird eine Zeit lang gewartet, bevor zum Zustand CLOSED übergegangen wird, damit alle eventuell noch vorhandenen Pakete, die einen neuen Aufbau irritieren könnten, verschwunden sind.

Tabelle 3-2: Zustandsabfolge des MC-Protokolls (Sicht der E2B-Bridge)

E2B-Bridge:		
CLOSED → LISTEN	-	-
Host Computer:		
LISTEN → SYN_RCVD	Ein Paket mit gesetztem SYN-Bit und beliebiger Sequenznummer (x) wird empfangen.	Ein Paket mit gesetztem SYN- und ACK -Bit wird zurückgeschickt. Bei der Sequenznummer (y) ist darauf zu achten, dass sie nicht mit alten noch vorhandenen Paketen verwechselt werden kann. Die ACK-Nummer soll das empfangene SYN-Paket bestätigen (ack = x + 1).
SYN_RCVD → ESTABLISHED	Ein ACK-Paket (ack = y + 1) und (seq = x + 1) wird empfangen.	-
ESTABLISHED → LAST_ACK	Ein FIN-Paket wird empfangen (seq = z)	Die Empfangs- und Sendepuffer werden gelöscht und ein Paket mit gesetzten FIN- und ACK-Flags wird zurückgeschickt (seq = u) und (ack = z + 1).
LAST_ACK → CLOSED	Ein ACK-Paket mit der ACK-Nummer u + 1 wird empfangen.	-

hen, falls prioritäre Daten übermittelt werden, und kann benutzt werden, um HCI-Kommandos mit höherer Priorität als ACL-Daten zu übermitteln. Das nächste Bit ist das *ACK*-Flag, welches gesetzt wird, falls das ACK-Feld gültige Daten enthält. Bit 6 ist das *RST*-Flag, welches im Fall eines Rejects gesetzt ist. Bit 7 (*SYN*-Flag) wird für den Verbindungsaufbau benutzt. Das letzte Bit im Type-Feld ist das *FIN*-Flag, welches für den Verbindungsabbau verwendet wird.

- *SEQ* (1 Byte):

Das Feld für die Sequenz-Nummer ist 1 Byte gross. Dies ermöglicht eine Windowgrösse von maximal 128. Eine grössere Windowgrösse bewirkt im Allgemeinen auf Grund der HCI-Flusskontrolle keinen Geschwindigkeitsvorteil mehr (siehe Kapitel 3.3.5). Als Abgrenzung nach unten haben wir uns entschieden, auf alle Fälle mindestens ein Byte für die Nummern zu verwenden, da in diesem Fall ein übersichtlicher und schneller Programmcode möglich ist.

- *ACK* (1 Byte):

Die Bestätigung kann entweder huckepack mit einem normalen Paket geschehen, oder mit einem ACK-Paket, bei welchem die Nutzdatenlänge auf 0 gesetzt wird. Dabei wird mit einer ACK-Nummer (A) die Sequenz-Nummer des nächsten erwarteten Pakets kommuniziert. Dabei werden alle ausstehenden Pakete bis zur SEQ-Nummer A - 1 bestätigt. Somit fallen verlorene ACK-Pakete nicht ins Gewicht, da mit dem nächsten erhaltenen ACK-Paket das alte auch bestätigt wird.

- *WIN* (1 Byte):

Das Window-Feld wird gebraucht, um dem Kommunikationspartner mitzuteilen, wie viele freie Empfangspuffer noch vorhanden sind. Wie im Kapitel 3.3.5 und bei der Erklärung des SEQ-Felds erläutert, befindet sich die Windowgrösse im Bereich 0 - 128.

- *LENGTH* (2 Bytes):

Das Grössen-Feld enthält die Länge der MC-Nutzdaten im Netzwerk-Byte-Format. Momentan ist die Grösse auf maximal 1024 Bytes beschränkt, in späteren Versionen kann sie aber bis auf die maximale HCI-Paketgrösse vergrössert werden.

### 3.3.5 Flusskontrolle und Management der Windowgrössen

#### 3.3.5.1 Evaluation

Bei der Evaluation des Sliding Window haben wir uns auf folgende Eckdaten abgestützt (siehe Tabelle 3.3.5.1):

#### 3.3.5.2 Dimensionierung der Grösse des Windows

Aus Messungen am bestehenden Bluez-Stack haben wir herausgefunden, dass die Toleranz für zusätzliche Verzögerungen auf HCI-Ebene ungefähr eine Sekunde beträgt. Um sicherzustellen, dass die Netzwerkverzögerung auch bei grossen Distanzen und widrigen Netzwerkbedingungen möglichst unter einer Sekunde liegt, ist der Einsatz eines Sliding Windows unabdingbar.

erwartete Netzwerkverzögerungen:	< 1 ms (LAN), ~ 1 ms (Gebäude) ~ 1 sec (Internet)
erwartete Datenraten:	~ 7kbytes/s (ROK), ~ 1Mbytes/s (Bluetooth)
Paketgrösse:	- 762 Bytes (ROK)
NUM_OF_PAKETS:	1 - 8 (je nach ROK-Variante)

*Tabelle 3-3: Eckdaten zum Design des Sliding Windows:*

*Die erwarteten Datenraten sind bei den eingesetzten Bluetooth-Modulen, welche mit 57600 Baud arbeiten, auf etwa 7kb/s beschränkt. Beachtet werden muss auch die Flusskontrolle der HCI-Schicht: Bei den eingesetzten Modulen ist die Anzahl der maximal ausstehenden Pakete in der Grössenordnung 1 - 8.*

Trotzdem muss beachtet werden, dass die Flusskontrolle auf HCI-Schicht die Anzahl der ausstehenden Pakete stark limitiert. Zwar ist der Maximalwert der ausstehenden Pakete in der Bluetooth-Spezifikation [1] auf 255 festgelegt, der Wert liegt aber bei den aktuellen BT-Modulen bei 1 - 8. Bei gewissen HCI-Sequenzen sind die HCI-Kommandos auch voneinander abhängig. So muss zum Beispiel beim Initialisieren des Bluetooth-Moduls nach jedem HCI-Kommando zuerst auf eine Antwort gewartet werden, bevor das nächste geschickt werden kann.

Dies sind die Gründe, weshalb wir uns entschieden haben, eine maximale Windowgrösse von 128 zu definieren.

### 3.3.5.3 Implementierung

Grundsätzlich teilt ein Gerät dem Kommunikationspartner beim Senden jedes Pakets mit, für wie viele Pakete es noch Platz in seinem Eingangspuffer hat. Dies geschieht mit dem Header-Feld WIN (siehe Kapitel 3.3.4). Somit ist es jedem Gerät überlassen, wie ausgefeilte die Empfangsmechanismen sind, die es zur Verfügung stellt. Dies fördert auch eine stufenweise Entwicklung, so kann mit einem Empfangspuffer begonnen werden und dann auf mehrere Puffer übergegangen werden, ohne dass die Gegenstelle neu programmiert werden muss.

Folgende Implementationen sind möglich: Idle-Repeat-Request als das einfachste Protokoll, dabei ist der Wert im Headerfeld WIN entweder 1 oder 0. Ebenso ist Sliding Window mit einem Empfangspuffer möglich (WIN = 1 oder 0), oder Sliding Window mit mehreren Empfangspuffern (WIN = 0 - Anzahl der Empfangspuffer).

Beachtet werden muss, dass die Verbindung blockieren kann, wenn ein Gerät keinen Empfangspuffer mehr frei hat und WIN auf 0 setzt. Das Problem ist, wie dem Kommunikationspartner mitgeteilt werden kann, dass wieder Empfangspuffer frei sind. Dies kann mit einem Paket ohne Nutzdaten und einem WIN grösser als null geschehen. Da dieses Paket aber verloren gehen kann, muss entweder dieses Paket wiederholt werden, bis das andere Gerät ein weiteres Paket schickt, oder aber das andere Gerät fragt immer wieder nach, ob es weitere Pakete schicken kann. In

unserem Protokoll ist dies folgendermassen gelöst: Jedes Gerät versucht immer ein Paket mehr zu senden, als es eigentlich auf Grund des WIN-Feldes dürfte. Damit wird immer wieder versucht, ein Paket zu schicken und ein Blockieren der Verbindung ist nicht möglich.

### *3.3.6 Fehlermanagement*

Die zwei folgenden Fehlerszenarien müssen beachtet werden:

- **Halboffene Verbindungen:**

Halboffene Verbindungen können zum Beispiel nach einem Crash auftreten.

Entgegengewirkt wird halboffenen Verbindungen mit einem Keepalive-Timer. So wird nach einer gewissen Idle-Zeit beim Kommunikationspartner nachgefragt, ob dieser immer noch vorhanden ist. Falls nach einigen Versuchen immer noch keine Antwort zurückgeschickt wird, wird der Verbindungsabbau eingeleitet.

Während dem Verbindungsabbau muss ein weiteres Timeout implementiert werden, um zu verhindern, dass ein Gerät ewig auf die Antwort eines eventuell nicht mehr vorhandenen Kommunikationspartner wartet. Falls dieses Timeout anspricht, wird die Verbindung einseitig als beendet erklärt.

- **Zwei Protokoll-Server versuchen mit der selben E2B-Bridge eine Verbindung aufzubauen:**

Das Problem tritt auf, wenn eine E2B-Bridge bereits eine Verbindung mit einem Protokoll-Server aufgebaut hat und dann eine Verbindungsaufforderung (SYN-Paket) von einem anderen Protokoll-Server erhält.

Dieses Problem ist folgendermassen gelöst: Wenn eine E2B-Bridge bereits eine aktive Verbindung hat, ignoriert sie alle weiteren Verbindungsaufforderungen.

### *3.3.7 Implementationsdetails*

In diesem Kapitel werden die Probleme beschrieben, die sich während dem Implementationsprozess gezeigt haben.

#### *3.3.7.1 ACK Verlust*

Das Szenario ist das folgende: Computer A schickt ein Daten-Paket zum Computer B. B akzeptiert das Paket, führt das Empfangs-Window nach und schickt eine Bestätigung zu A zurück. Dieses Paket geht aber verloren. Nach dem Ansprechen des Timeouts sendet A das Daten-Paket nochmals. Dieses Paket liegt aber richtigerweise nicht mehr im Empfangs-Window von B, sodass es von B verworfen wird.

Um in diesem Szenario endlose Wiederholung zu vermeiden, ist es nötig, dass wenn Pakete ausserhalb des Empfang-Puffers empfangen werden, ein ACK-Paket mit der aktuellen ACK-Nummer zurückgeschickt wird.

### 3.3.7.2 Unterscheidung zwischen ACK-Paketen und leeren Paketen

Für ACK-Pakete gibt es keinen speziellen Typ, sondern ACK-Pakete sind Daten-Pakete ohne Nutzdaten. Da ACK-Pakete von der Gegenstelle nicht bestätigt werden, wird die SEQ-Nummer des letzten Nutzdaten-Pakets nochmals verwendet.

Dabei ist es wichtig, dass keine leeren Nutzdaten-Pakete geschickt werden dürfen.

Im folgenden Szenario kann es sonst zu einem Versagen des Protokolls kommen. Computer A schickt ein Paket P an Computer B, das aber auf dem Weg verloren geht. Falls nun Computer A ein ACK-Paket vor der erneuten Übertragung von P zu Computer B schickt, hält dieser das ACK-Paket für ein leeres Datenpaket und verwirft danach das erneut übertragene Paket P, da er es für ein Duplikat hält.

## 3.4 Erweiterbarkeit des MC-Protokolls

Ein wichtiger Punkt beim Design des MC-Protokolls war die Erweiterbarkeit des Protokolls.

So sind mehrere Realisierungen der Flusskontrolle möglich. Angefangen beim simplen Idle-Repeat-Request bis zum Sliding Window mit mehreren Empfangspuffern. Diese Implementierungsvarianten können völlig transparent implementiert werden, das heisst, die Gegenstelle muss nicht über eine Implementationsänderung informiert werden. Somit wird sichergestellt, dass E2B-Bridge und Host-Software unabhängig voneinander entwickelt werden können. Ebenso ist ein schrittweises Implementieren und Testen möglich, was die Entwicklung vereinfacht.

Ebenso wurde an allfällige Erweiterungen in der Zukunft gedacht. Da immer ganze HCI-Pakete mit einem MCP-Paket übertragen werden, ist es auf Seite der E2B-Bridge einfach möglich, eigene HCI-Pakete in den Datenstrom einzugliedern. Das heisst, falls in der Zukunft der L2CAP-Layer doch noch auf die E2B-Bridge ausgelagert wird, können die L2CAP-Daten einfach zwischen den MCP-Nutzdaten zum Bluetooth-Modul geschickt werden.

Mit dem Definieren eines Typen- und Version-Feldes im MCP-Header ist es auch möglich, Anpassungen am Protokoll vorzunehmen. So können z.B. zusätzliche Daten zwischen der E2B-Bridge und dem Host-Computer ausgetauscht werden. Ein mögliches Szenario ist, dass der Host-Computer die E2B-Bridge mit einem speziellen Paket darüber informiert, dass die Baudrate der UART-Verbindung geändert werden soll.



# 4

## *Implementation Software*

Dieses Kapitel beschreibt die Implementation des Softwarepakets *Netty*, welches auf dem Host-Computer laufen muss, damit mit E2B-Bridges kommuniziert werden kann. Dabei wird im Kapitel 4.1 untersucht, welche Funktionalität *Netty* implementieren muss. In den Kaptitel 4.2 und 4.3 wird dann erläutert, wie *Netty* am besten in die bestehende Software integriert werden kann. Abschliessend wird in Kapitel 4.4 auf die eigentliche Implementation eingegangen.

### *4.1 Spezifikationen*

Aus den Einsatzmöglichkeiten (siehe Kapitel 1) der E2B-Bridge können folgende Spezifikationen für die Software abgeleitet werden.

- Fehlerfreie Verbindung :

Das *MC-Protokoll* muss vollständig implementiert werden, damit eine fehlerfreie Verbindung mit der Bridge gewährleistet werden kann.

- Verbindungsmanagement :

Da von einem Kontrollcomputer aus mehrere Bridges überwacht werden sollen, muss die Software mehrere verschiedene Verbindungen verwalten können. Ebenso muss sie in der Lage sein, während dem Betrieb eine neue Verbindung aufzubauen und wieder abzurechnen.

Ebenso soll ein möglichst grosser Synergieeffekt genutzt werden, indem bereits vorhandene Software bestmöglich weiter verwendet wird.

### *4.2 Vorhandene Software - Bluez*

Als Grundlage für diese Arbeit wurde der Bluetooth-Stack *Bluez* [3] gewählt. Dies weil die Software als Open-Source zur Verfügung steht und mit dem Linux-Kernel schon standardmässig mitgeliefert wird, was auch auf eine genügende Stabilität hinweist.

### 4.2.1 Allgemeines Funktionsprinzip

BlueZ implementiert die wichtigsten Befehle des *Bluetooth Hosts* gemäss der Bluetooth Spezifikation [1].

Die für das Betriebssystem Linux entwickelte Software besteht aus den Managementtools im Userbereich und dem hardwarenahen Teil, der mittels Kernel-Modulen realisiert ist (siehe Abbildung 4-1).

Die Kommunikation zwischen den beiden Bereichen findet durch ein Socket-Interface statt, auf welches in Kapitel 4.2.2 genauer eingegangen wird.

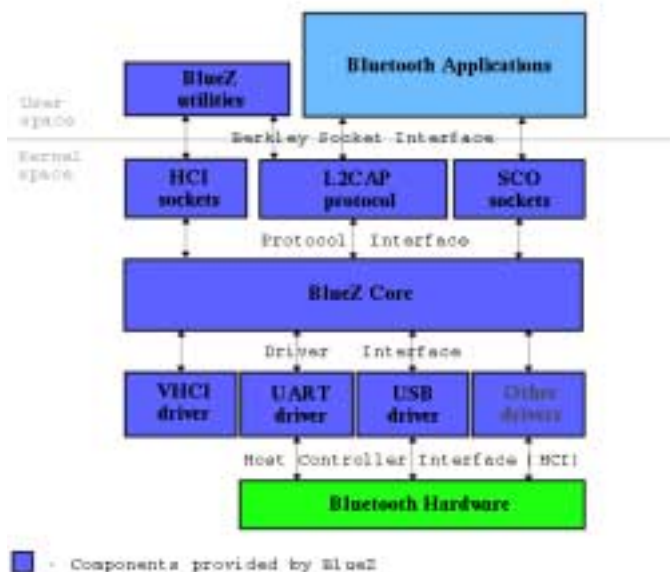


Abbildung 4-1  
Struktur des  
BlueZ-Stacks:  
Die Aufteilung des  
BlueZ-Stacks in die  
verschiedenen Module.

Der Kernel-Bereich wiederum ist wie folgt gegliedert:

- Geräte-Treiber :

Diese Module auf der untersten Schicht stellen die Schnittstelle zu den verschiedenen unterstützten *Host Controller Interfaces* dar. Unterstützt werden bis jetzt UART und USB. Ebenso gibt es für Testzwecke ein virtuelles Bluetooth-Modul (VHCI). In diesem wird die wichtigste Funktionalität eines realen Bluetooth-Moduls nachgebildet, damit die höheren Schichten getestet werden können, ohne dass man im Besitz eines Bluetooth-Moduls sein muss.

- BlueZ-Core :

In diesem Modul werden die HCI-Kommando-Pakete generiert und in der richtigen Reihenfolge dem Geräte-Treiber übergeben. Somit findet hier die eigentliche Umsetzung zwischen den Kommandos der oberen Schichten und den HCI-Kommandos statt.

- Protokoll-Schicht :

In dieser Schicht werden die verschiedenen Socket-Protokolle implementiert. Im Moment sind dies die Protokolle *RAW*, *L2CAP* und *SCO*. *RAW* wird für die

Administration der Bluetooth-Module gebraucht, während *L2CAP* und *SCO* für Datenübertragungen benutzt werden.

### 4.2.2 Schnittstelle zur Applikation: Socket Interface

Die Kommunikation zwischen Kernel-Bereich und User-Bereich und somit auch zwischen dem *Bluez-Stack* und einer User-Applikation findet über das *Berkeley Socket Interface* statt.

Somit wird der *Bluez-Stack* vollständig gekapselt und der Applikationsprogrammierer muss nichts von der Funktionsweise des Stacks wissen, sondern kann wie bei jeder anderen Netzwerkverbindung via Sockets auf das Bluetooth-Modul zugreifen.

#### 4.2.2.1 Daten-Verbindungen

Mit dem folgenden Beispielcode kann eine ACL-Datenverbindung, ausgehend vom Modul mit der Adresse 01:02:03:04:05:06, aufgebaut werden.

Zuerst wird ein Bluetooth Socket vom Typ *SEQPAKET* und dem Protokoll *L2CAP* geöffnet:

```
int s = socket(PF_BLUETOOTH, SOCK_SEQPAKET, BTPROTO_L2CAP);
```

Dann wird die lokale Adresse festgelegt:

```
char bluetooth_address[] = "01:02:03:04:05:06";
local_addr.l2_family = AF_BLUETOOTH;
baswap(&local_addr.l2_bdaddr, strtoba(blueetooth_address));
```

Und schliesslich mit einem `bind()` und `connect()` die Verbindung aufgebaut:

```
bind(s, (struct sockaddr*)&local_addr, sizeof(local_addr));
connect(s, (struct sockaddr*)&remote_addr, sizeof(remote_addr));
```

Darauf können nun mit `send()` Daten geschrieben werden:

```
send(s, data, data_size, 0);
```

#### 4.2.2.2 RAW-Sockets

Für Verwaltungszwecke kann über RAW-Sockets direkt auf die Bluetooth-Module zugegriffen werden. Dies ist im Prinzip auf zwei verschiedene Arten möglich. Erstens mittels *IOCTL-Aufrufen* oder indem die HCI-Kommandos direkt auf den RAW-Socket geschrieben werden.

Wie aus folgenden Codestücken ersichtlich ist, stellt *Bluez* dem Applikationsprogrammierer in der Bluetooth Library (`libbluetooth`) einige Hilfsfunktionen zur Verfügung, welche die Programmierung stark vereinfachen.

Weiterführende Code-Beispiele können in `hciconfig.c` und `l2test.c` gefunden werden (siehe Anhang A.1).

- IOCTL-Aufrufe:

Öffnen eines RAW-Sockets:

```
ctl = socket(AF_BLUETOOTH, SOCK_RAW, BTPROTO_HCI);
```

Dann muss festgelegt werden, auf welches Gerät zugegriffen werden soll.

```
dr.dev_id = device_number;
```

Abschliessend findet der IOCTL-Aufruf statt, hier als Beispiel mit der Aufforderung, das Gerät in den Authentifikationsmodus zu versetzen (HCISETAUTH):

```
ioctl(ctl, HCISETAUTH, (unsigned long)&dr);
```

- Schreiben und Lesen von einem RAW-Socket:

Die folgende Library Funktion führt die Funktionen `socket()` und `bind()` aus:

```
s = hci_open_dev(device_number);
```

Die folgenden zwei Beispiele zeigen die beiden Möglichkeiten, wie danach mit dem Bluetooth-Modul kommuniziert werden kann.

- Versenden von Kommandos (PC → Bluetooth-Modul):

Auf diesen Socket können nun direkt HCI-Kommandos geschickt werden, wobei für das Zusammenstellen des HCI-Pakets die Funktion `hci_send_cmd` benützt werden kann:

```
hci_send_cmd(s, MY_OGF, MY_OCF, parameter_length,  
(void *)&parameter);
```

- Versenden von Requests (PC ↔ Bluetooth-Modul):

Die folgende Funktion `hci_send_req()` kann verwendet werden, falls die Antwort des Bluetooth-Modul auf das HCI-Kommando auch ausgewertet werden soll. Die Funktion sendet das HCI-Kommando, setzt den Empfangsfilter entsprechend dem Kommando und wartet auf die Antwort, welche als `request.rparam` (by reference) zurückgegeben wird.

```
request.ogf = MY_OGF;  
request.ocf = MY_OCF;  
request.cparam = NULL;  
request.clen = 0;  
request.rparam = &request_parameter;  
request.rlen = MY_RESPONSE_LENGTH;  
  
hci_send_req(s, &request, timeout);
```

### *4.2.3 Konfiguration der Schnittstelle zum Host Controller Transport Layer*

Falls *USB* als Medium zwischen Host und Host Controller benützt wird, muss nur der USB Treiber von Linux für den neuen Gerätetyp konfiguriert werden. Danach

wird das Bluez-USB-Modul automatisch geladen und konfiguriert, sobald ein neues Bluetooth-Gerät mit dem Computer verbunden wird.

Werden hingegen Bluetooth Module mit *UART* Schnittstelle verwendet, muss der Benutzer den *Bluez-Stack* über Konfigurationsänderungen informieren, da ein automatisches Erkennen von neuen Geräten bei dieser Schnittstelle nicht vorgesehen ist. Für diese Konfiguration betreffend Geschwindigkeit, Art der Flusskontrolle und Typ des Moduls stellt der *Bluez-Stack* das Hilfsprogramm *hciattach* (siehe Anhang A.1) zur Verfügung.

#### 4.2.3.1 *Tools zur Konfiguration des Bluetooth-Moduls*

Mit dem Hilfsprogramm *hciconfig* (siehe Anhang A.2.1) lassen sich die meisten Funktionen zur Administration und Konfiguration von Bluetooth-Modulen ausführen. Die wichtigsten Funktionen sind das Initialisieren eines Moduls, das Herunterfahren eines Moduls, sowie das Ausführen von Inquiries.

## 4.3 *Integrationskonzept der neuen Software*

In diesem Kapitel soll untersucht werden, wie sich die neue Software am besten in die alte integrieren lässt. Dabei wird zuerst beschrieben, welches die in der Theorie beste Lösung ist, dann wird die in der Praxis gewählte Lösung erklärt.

### 4.3.1 *Konzept im Kernel-Bereich*

Konzeptionell am schönsten ist es, die neue Funktionalität auf der Gerätetreiber-Schicht des Bluez-Stacks einzufügen. Es wird ein neues Kernel-Modul [21] [22] programmiert, das die Daten von der nächst höheren Schicht entgegennimmt und sie über das Netzwerk zur *E2B-Bridge* sendet (siehe Abbildung 4-2).

Während eine starke Verankerung der neuen Software im *Bluez-Stack* konzeptionell anzustreben ist, hat diese Lösung aber einige Nachteile. Damit das neue Software-Modul vom User-Bereich aus konfiguriert werden kann, müssen neue IOCTL-Aufrufe implementiert werden, diese Informationen in geeigneten Datenstrukturen gespeichert und gleichzeitig an das Kernel-Modul weitergegeben werden. Dies ist nur mit grossen Anpassungen am Bluez-Code möglich. Ein weiteres Problem ist, dass die Netzwerkprogrammierung im Kernel-Bereich komplex und nur spärlich dokumentiert ist [23] [24].

### 4.3.2 *Konzept im User-Bereich*

In einem zweiten Konzept wird die neue Software weitgehend vom *Bluez-Stack* gekapselt. Als Schnittstelle zum *Bluez-Stack* werden *Pseudo-Terminals* [29] [30] benutzt. Diese Terminals werden von Linux standardmässig zur Verfügung gestellt und funktionieren folgendermassen: Ein *Pseudo-Terminal* besteht aus zwei Devices, einem Master (*/dev/ptyX*) und einem Slave (*/dev/ttyX*). Beide Devices funktionieren wie normale Terminals, wobei alle Daten, die in das eine Device geschrieben wird, am anderen wieder gelesen werden können. Dabei kann das *ttyX* wie ein serielles Gerät behandelt werden. Diese Funktionalität wird von Linux standardmässig zur Verfügung gestellt und kann einfach in Programme implementiert werden. Dank

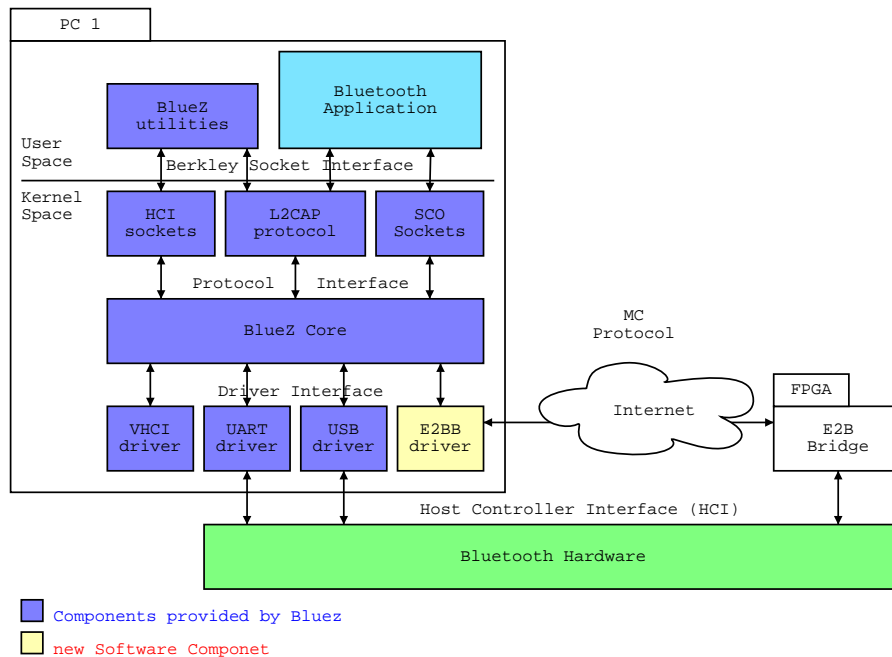


Abbildung 4-2

Struktur der Software gemäss dem Kernel-Bereich Konzept:  
Der E2BB-Treiber wird als Kernel-Modul an der Stelle eingefügt, der vom BlueZ-Stack für andere Treiber vorgesehen wurde.

dieser Trennung von *Bluez-Stack* und neuer Software können die beiden Pakete getrennt voneinander entwickelt und zukünftige Änderungen des *Bluez-Stacks* einfach übernommen werden. Um auch im Bereich UART-Treiber Konfiguration eine möglichst gute Kapselung zu erreichen, wird die Konfiguration weiterhin mit den *Bluez-Utilities* vorgenommen, die über einen Wrapper angesprochen werden.

Mit diesem Design kann der grösste Teil der Software im User-Bereich implementiert werden, was die Netzwerkprogrammierung stark vereinfacht [25] [26] [27] [28]. Eine Übersicht über die User-Bereich Variante wird in Abbildung 4-3 gezeigt.

Ein mögliches Problem in diesem Design ist, wie eine Änderung der UART-Geschwindigkeit der Bridge gemeldet werden kann, da das Netzwerkmodul diese Änderung durch das *Pseudo-Terminal* nicht bemerkt. Somit muss entweder die Bridge das HCI-Kommando, welches bei der Änderung der Geschwindigkeit zum Bluetooth-Modul geschickt wird (`HCISET_UART_BAUDRATE`), erkennen und entsprechend reagieren, oder das Netzwerk-Modul von *Netty* muss diese Aufgabe übernehmen und die Änderung mittels eines speziellen *MCP-Pakets* der Bridge mitteilen.

## 4.4 Konzept des Softwarepakets Netty

Da das Konzept mit der Implementation im User-Bereich mehrere Vorteile bietet, wurde das Softwarepaket *Netty* gemäss dem User-Bereich Konzept implementiert. Der ausschlaggebende Vorteil ist die gute Kapselung der neuen Software. So kann

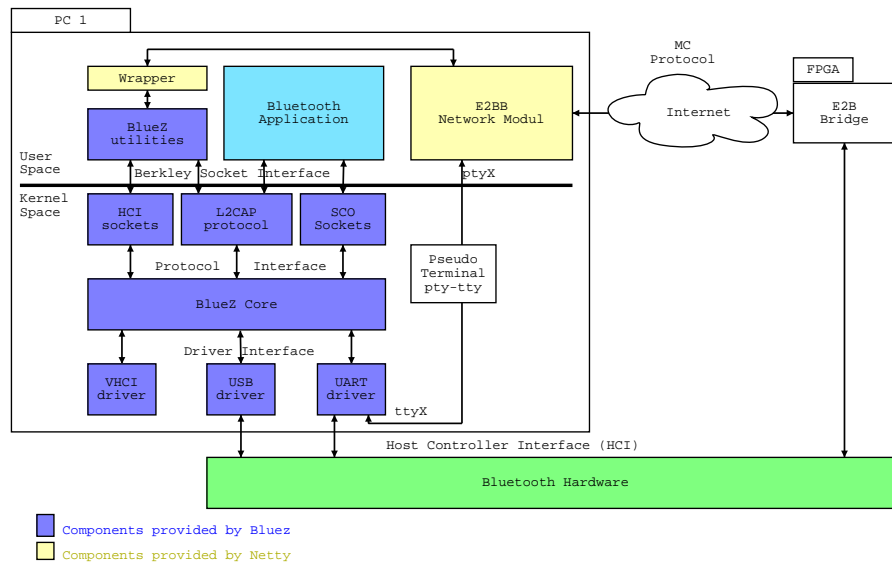


Abbildung 4-3

Struktur der Software gemäss dem User-Bereich Konzept: Die Anbindung geschieht mittels eines Pseudo-Terminals auf HCI-Ebene und mit einem Wrapper für die Konfiguration.

Die Netzwerkprogrammierung findet vollständig im User-Bereich statt.

Netty mit geringen Änderungen an neue BlueZ-Versionen angepasst werden. Ein weiterer gewichtiger Pluspunkt ist auch die einfachere Handhabung der Netzwerkprogrammierung im User-Bereich. Dies ermöglicht ein übersichtliches und strukturiertes Design der gesamten Software.

Die Software besteht aus zwei Modulen, wobei das Netzwerkmodul für die Kommunikation zwischen *Pseudo-Terminal* und Netzwerk zuständig ist und der Wrapper für die richtige Konfiguration des *BlueZ-Stacks* zuständig ist (siehe Abbildung 4-3). Dabei werden beide Module je als eigene Prozesse realisiert.

#### 4.4.1 Netzwerk-Modul

Das Netzwerk-Modul wurde bewusst mit nur einem Prozess implementiert, damit auf fehlerträchtige Interprozess-Kommunikation verzichtet werden kann. Trotzdem soll aber ein Netzwerk-Modul Verbindungen zu mehreren *E2B-Bridges* verwalten können, die sich je in einem anderen Zustand befinden können (siehe 3.3.3). Um zu verhindern, dass eine Verbindung blockiert wird, wenn eine andere auf Daten wartet, wurde ein ereignisgesteuertes Design gewählt. Es gibt eine Hauptschleife im Programm, bei der jede Verbindung diejenigen Events eintragen kann, auf die es gemäss ihrem Zustand warten will. Die Hauptschleife wartet dann auf das Eintreten dieser Ereignisse und ruft die entsprechenden Callback-Handler auf (siehe Abbildung 4-4). Mögliche Ereignisse und Aktionen sind in Tabelle 4.4.1 zusammengefasst.

Ereignis	→Aktion
Benutzer Interaktion	Eine Verbindung zu einer neuen <i>E2B-Bridge</i> soll aufgebaut oder eine bestehende Verbindung soll abgebrochen werden.
Neue Daten vom <i>Pseudo-Terminal</i>	Die Daten werden zur entsprechenden <i>E2B-Bridge</i> weitergeleitet.
Neue Daten vom Netzwerk	Reaktion gemäss MC-Protokoll, eventuell vorhandene Nutzdaten müssen zum entsprechenden <i>Pseudo-Terminal</i> weitergeleitet werden.
Netzwerk Timeout: Es wurde für ein gesendetes Paket kein Acknowledge empfangen.	Das entsprechende Paket muss nochmals gesendet werden.

Tabelle 4-1: Mögliche Ereignisse, die während der Hauptschleife des Programms auftreten können und die daraus resultierende Aktionen

#### 4.4.2 Prozessfluss von Netty

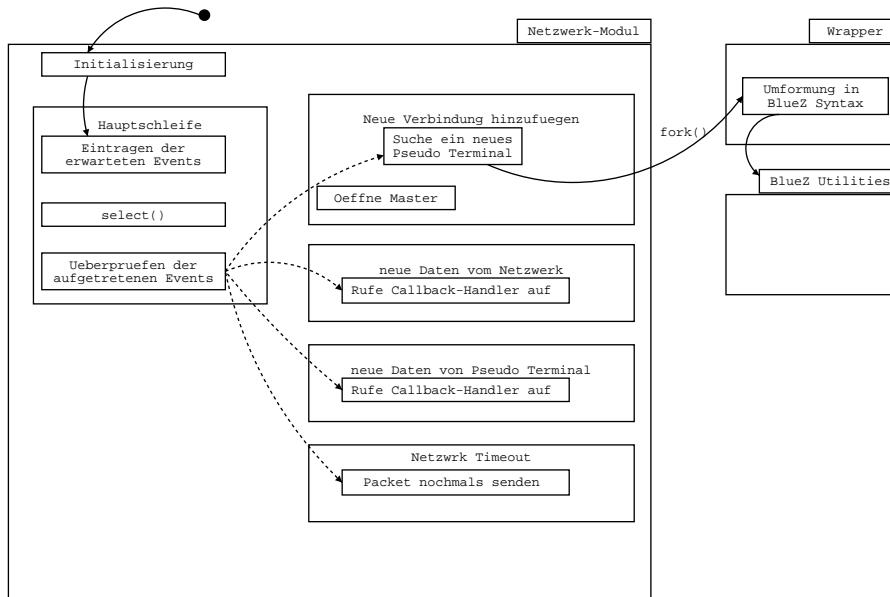


Abbildung 4-4 Prozessfluss von Netty mit der Hauptschleife und den vier möglichen Events die auftreten können: 1. User Interaktion (beispielhaft wird der Aufbau einer neuen Verbindung gewählt) 2. Neue Daten vom Pseudo-Terminal 3. Neue Daten vom Netzwerk 4. Ein Netzwerk Timeout

Beim Starten von *Netty* wird das Programm initialisiert und beginnt mit der Abarbeitung der Hauptschleife. Da noch keine Verbindungen aktiv sind, ist das einzige erwartete Event eine Aufforderung des Benutzers, eine neue Verbindung aufzubauen. Falls diese Aufforderung eintritt, ist der erste Schritt das Suchen eines freien *Pseudo-Terminals*. Das Master-Device wird vom Netzwerk-Modul verwendet. Das Slave-Device hingegen wird vom neu kreierten Wrapper-Prozess durch die *Bluez-Utilities* dem *Bluez-Stack* übergeben (siehe Abbildung 4-4).

### 4.4.3 Datenstrukturen

Um den in Kapitel 4.4.2 beschriebenen Prozessfluss implementieren zu können, sind folgende Datenstrukturen notwendig.

Auf Ebene Netzwerk-Modul wird eine *sortierte Liste mit allen Netzwerk-Timeouts* verwaltet, damit jeder `select()`-Aufruf mit der Timeout-Zeit des nächsten auftretenden Netzwerk-Timeout programmiert werden kann. Ebenso wird eine *Liste mit allen Verbindungen* zu E2B-Bridges gespeichert.

Für jede dieser Verbindungen werden noch weitere Zustandsinformationen gespeichert. Einerseits sind dies Daten über die *Gegenstelle* der Verbindung (wie IP-Adresse und Port). Andererseits sind es Daten über den *Zustand der MCP Zustandsmaschine* und *MCP-Verbindungsdaten* (wie aktuelle Sequenznummern, Fenstergrößen, Zustand des OUT- und IN-Stacks). Ebenso müssen *HCI-Fragmente*, die vom Pseudo-Terminal gelesen werden, zwischengespeichert werden, bis ein ganzes HCI-Paket zusammen ist (siehe Kapitel 3.3).

### 4.4.4 Implementationsdetails

In diesem Kapitel werden einige wichtige Implementationsdetails aufgezeigt und erklärt.

#### 4.4.4.1 Flusskontrolle

Um einen Überlauf des eigenen Netzwerk-Stacks und des Stacks der Gegenstelle zu vermeiden, muss auch im Netzwerk-Modul eine Flusskontrolle integriert sein. Es muss sichergestellt werden, dass das Lesen vom Pseudo-Terminal eingestellt wird, sobald ein Stack voll ist, und wieder aufgenommen wird, sobald wieder Platz vorhanden ist. Dies wird in *Netty* erreicht, indem das Event "neue Daten vom Pseudo-Terminal" aus dem Event-Handler der Hauptschleife entfernt wird, sobald ein Stack voll ist. Die Überprüfung, ob das Event wieder eingefügt werden darf, muss nach jedem empfangenen Paket wieder vorgenommen werden, da sich der Stack der Gegenseite oder der eigene durch dieses Paket verändert haben kann.

Auch berücksichtigt werden muss der Spezialfall, wenn mit einem Lesebefehl vom Pseudo-Terminal mehr als ein HCI-Kommando gelesen wird, aber der Stack nur noch Platz für ein Paket bietet. Dies kann nicht verhindert werden, da die Grösse der Kommandos nicht von vornherein bekannt ist und das Pseudo-Terminal einen Bytestream ohne Kommandogrenzen zur Verfügung stellt.

Falls dieser Spezialfall auftritt, muss das Kommando, welches noch nicht geschickt werden kann, zwischengespeichert werden. Ebenso muss dem Event-Handler der

Hauptschleife mitgeteilt werden, dass vor den neuen Daten vom Pseudo-Terminal die Daten im Puffer zur E2B-Bridge weitergereicht werden müssen.

#### *4.4.4.2 Debugging-Konzept*

Für die Fehlersuche wurde ein mehrstufiges Debugging-Konzept implementiert. So kann durch Konfiguration des Debugging-Levels die Anzahl der vom Programm generierten Meldungen beeinflusst werden. Dabei steht der Level 0 für keine Meldungen und Level 9 für alle Meldungen. Des Weiteren kann getrennt bestimmt werden, wo diese ausgegeben werden sollen. Dabei ist es sowohl möglich, eine Datei anzugeben, als auch StdOut und StdErr zu verwenden. Somit wird eine grösstmögliche Flexibilität bei der Fehlersuche erreicht. Beachtet werden muss aber, dass die ungepufferte Ausgabe von Debugging-Meldungen stark ressourcenhungrig ist und somit bei schwach ausgestatteten Systemen durch zu hohe CPU- und IO-Last zu Problemen führen kann.

Als weiteres Feature, um die Fehlertoleranz des MC-Protokolls zu testen, ermöglicht es *Netty* eine Droprate einzustellen, die bei jedem ausgehenden Paket zufällig gemäss der eingestellten Droprate entscheidet, ob das Paket geschickt wird oder verworfen werden soll. Diese Funktion kann verwendet werden, um Netzwerkverluste zu simulieren.

#### *4.4.4.3 GUI-Schnittstelle*

Die Konfiguration von *Netty* durch ein GUI ist im Design bereits vorgesehen. Da als Schnittstelle für die User Interaktion sowohl StdIn als auch ein Pseudo-Terminal vorgesehen wurde, ist es möglich ein GUI in *Netty* zu integrieren, das auch in einem eigenen Prozess ablaufen kann und somit getrennt von den übrigen *Netty*-Modulen entwickelt werden kann. Die Interprozess-Kommunikation kann dann mittels des Pseudo-Terminals geschehen.

In einem ersten Schritt wurde eine einfache Version eines GUI mit *qt* [10] implementiert, das es dem Benutzer erlaubt, die Verbindungsdaten in zwei Formularen einzugeben, ohne dass er sich lange Kommandozeilenoptionen merken muss. Im ersten Formular können die Netzparameter der Gegenstelle und die Bluetooth-Parameter des daran angeschlossenen Moduls eingegeben werden. Im zweiten können vom versierten Benutzer zusätzliche Optionen eingestellt werden (siehe Abbildung 4-5).



*Abbildung 4-5*  
*Screenshots vom GUI:*  
*Das erste Formular mit den Netzwerk- und Bluetooth-Parametern, das*  
*zweite mit den optionalen Einstellungen.*



# 5

## *Implementation Hardware*

In diesem Kapitel wird die Hardware der E2B-Bridge beschrieben. Kapitel 5.1 zeigt verschiedene Architektur-Alternativen auf, während in Kapitel 5.2 die von uns gewählte Architektur gezeigt wird. Die Kapitel 5.3 bis 5.4 erklären daraufhin die Details unseres Designs. Kapitel 5.5 ist schliesslich dem Aufbau unseres Testbenchs gewidmet.

### *5.1 Designspace*

In unserer Aufgabenstellung war vorgegeben, dass die Bridge auf einem FPGA-Board implementiert werden sollte. Es stand uns ein XSV-Board von der Firma XESS zur Verfügung, das am TIK schon vorhanden war. Es enthält einen Xilinx *Virtex XCV800*. Eine kurze Beschreibung dieses Boards ist in Kapitel 2.3.1 zu finden.

Auf dem FPGA-Board sind für den OSI-Layer 1 der beiden Interfaces (Ethernet resp. RS232) dedizierte Chips vorhanden, alles andere muss im FPGA implementiert werden. Es gibt sehr vielfältige Möglichkeiten, wie man die gewünschte Funktionalität realisieren kann. Es gibt zum einen die Möglichkeit, einen Prozessor-Core in unser Design zu integrieren und einen Teil der Bridge in Software zu programmieren. Für die Schnittstellen nach aussen braucht man dann lediglich noch eine virtuelle "Netzwerkkarte" und einen "UART-Baustein", die beide gemeinsam mit dem Prozessor auf dem FPGA integriert werden.

Auf der anderen Seite gibt es auch die Möglichkeit, eine vollständige dedizierte Hardware zu entwerfen, die dieselbe Funktionalität realisiert. Der ganze IP-Stack würde also in Blöcke aufgeteilt, die nebeneinander gleichzeitig auf dem FPGA laufen.

Beim Entscheid für eine dieser Varianten muss man besonders auf die Algorithmen und Daten achten, die zu bearbeiten sind.

Einige gute Richtlinien mit Kriterien für die Entscheidung, ob etwas in Hardware oder in Software implementiert werden sollte, gibt das Script der VLSI I-Vorlesung

[6] auf Seite 42ff. Demnach gibt es in unserem Fall folgende Argumente für eine Lösung mit einem Prozessor-Core:

- Es ist *HW/SW-Codesign* möglich, d.h. die Teile, die in Hardware effizienter zu entwickeln sind, werden in Hardware gebaut, die anderen werden im Prozessor-Core in Software implementiert. Dies bietet ausserdem den Vorteil, dass Hardware und Software gleichzeitig parallel entwickelt werden können, was eine kürzere Gesamt-Entwicklungszeit ermöglicht.
- Bei guter Partitionierung ist eine Lösung mit Prozessor-Core sehr viel flexibler, da die Software viel einfacher angepasst und ausgetauscht werden kann, als dies mit Hardware möglich ist.
- Für den IP-Stack, der den komplexesten Teil der Schaltung darstellt, kann eine vorhandene Software-Lösung verwendet werden.

Es ist auf alle Fälle ein sehr interessanter Ansatz, einen *Computer auf einem FPGA* zu bauen. Den Vorteilen stehen aber auch einige Nachteile gegenüber:

- Ein solches Design benötigt sehr viel Platz auf dem FPGA. Auf dem Board ist zwar ein grosser FPGA vorhanden, jedoch muss bei einer eventuellen späteren Anfertigung von vielen Exemplaren der Bridge für jede auch wieder ein grosser FPGA/ASIC genommen werden, was sich natürlich markant auf den Preis auswirkt.
- Eine Architektur mit Software auf einem Prozessor ist prinzipbedingt langsamer als eine in Hardware. Sie kann deshalb viel weniger Pakete pro Sekunde bearbeiten, da der Prozessor mit dem Verarbeiten der eintreffenden Daten nicht schnell genug nachkommt.

Dies hat uns herausgefordert, auch die reine Hardware-Variante zu untersuchen.

Sie bietet folgende Vorteile:

- Sie kann sehr viel schneller Daten verarbeiten.
- Sie ist (bei geeigneten Algorithmen) kompakter als eine Architektur mit einem General-Purpose-Prozessor.
- Sie ist damit im Allgemeinen auch Energie-effizienter.

Der Nachteil ist die eingeschränkte Flexibilität, da Software schneller und einfacher änderbar ist als Hardware. Wir waren jedoch der Meinung, dass dies durch die Vorteile aufgewogen wird. Dass schon eine andere Gruppe in ihrer Diplomarbeit [31] dabei war, eine Architektur mit einem Leon-Prozessor-Core [32] zu implementieren hat uns zusätzlich motiviert, den anderen Weg zu versuchen.

Wir suchten nach fertigen Blöcken, die wir für unser Design benutzen konnten. Dabei stiessen wir auf einen frei erhältlichen IP-Stack in VHDL [13], der an der Universität von Queensland in Australien entwickelt worden ist. Er war relativ vollständig und funktionierte in ersten Tests gut, so dass wir uns entschlossen, eine reine Hardware-Lösung zu bauen.

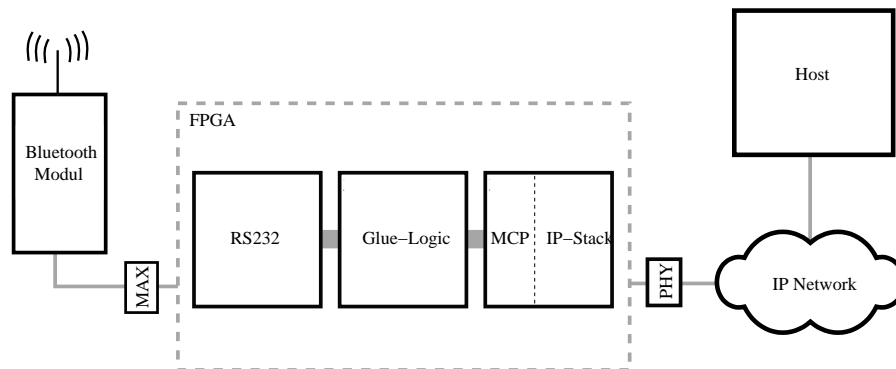


Abbildung 5-1  
Blockdiagramm des Designs

## 5.2 Architektur

Wie schon im vorhergehenden Kapitel erwähnt, haben wir einen fremdentwickelten IP-Stack als Building-Block in unser Design aufgenommen. Damit war es möglich, die Bridge auf dem IP-Protokoll aufzubauen, so dass Routing möglich ist und die E2B-Bridge auch an ein grösseres Netzwerk angeschlossen werden kann.

Wir untersuchten daraufhin, ob man den Stack noch um einen TCP-Layer erweitern könnte. Es zeigte sich aber bald, dass dies sehr kompliziert sein würde. Dies war der Grund, warum wir uns für ein eigenes, auf UDP aufsetzendes Protokoll mit einfacheren Strukturen und Abläufen entschieden haben, das MCP (siehe Kapitel 3.3).

Neben dem IP-Stack besteht die Bridge aus zwei weiteren Blöcken: Einem für das RS-232-Interface und einem mit Glue-Logik dazwischen. Er dient dazu, die Daten zwischen den beiden anderen Blöcken weiterzuleiten. Die MC-Protokollschicht haben wir in den IP-Stack integriert. Bild 5-1 zeigt ein Blockdiagramm des ganzen Designs.

Es gibt zwei weitgehend getrennte Datenflüsse: Einen vom Ethernet-Empfänger zum RS232-Sender, den anderen vom RS232-Empfänger zum Ethernet-Sender. Die einzige Verbindung dazwischen stellt der MCP-Block dar, wo die Sequenz- und Acknowledge-Nummern übergeben werden und der Zustand des MCP-Protokolls gespeichert wird.

Diese drei Blöcke mit ihrer Funktionalität und ihren Schnittstellen werden in den folgenden Kapiteln genauer beschrieben.

## 5.3 Der IP-Stack

### 5.3.1 Auswahl eines Designs

Der grosse Vorteil des von uns gewählten Stacks war, dass er mit Ethernet-Layer, ARP, IP, UDP sowie ICMP, wenn auch mit gewissen Einschränkungen, relativ

vollständig war.

Auf der XESS-Homepage [11] gab es auch noch eine andere Variante: Ein Design vom Bandung Institute of Technology [14]. Hier war aber lediglich ein MAC-Layer vorhanden, dieser zwar vollständiger als derjenige im Stack der Australier. Es hätte jedoch zuviel Aufwand gebraucht, um die ganzen fehlenden Blöcke zu ergänzen. Auch liessen sich die beiden Designs nicht einfach kombinieren, da die Interfaces verschieden waren. Deshalb war dieses zweite Design für uns nicht brauchbar und wir blieben bei demjenigen aus Queensland.

Layer	Eigenschaften	Einschränkungen
Ethernet	10Mbps Ethernet: Senden und Empfangen, CRC-Handling	Das Design muss an eine Vollduplex-Verbindung angeschlossen werden, da kein Carrier-Sensing und auch keine Collision-Detection implementiert sind.
ARP	Stellen von ARP-Requests bei unbekannter MAC-Adresse sowie Entgegennahme und Verarbeitung der zurückkommenden Replys. Beantworten von ARP-Requests anderer Geräte.	Die ARP-Tabelle ist zwar dynamisch, hat aber nur zwei Einträge. Ausserdem ist, nachdem ein ARP-Request gesendet wurde, der ganze Sender blockiert bis die Antwort zurückkommt oder ein Timeout ausgelöst wird.
ICMP	ICMP Echo Requests (Ping-Pakete) werden beantwortet.	Alle anderen ICMP-Pakete werden ignoriert.
IP	IP-Pakete (auch fragmentierte) können empfangen und gesendet werden.	Auf Empfangsseite müssen die Fragmente eines Paketes in der richtigen Reihenfolge ankommen, da sie ansonsten verworfen werden.
UDP	Sender und Empfänger	UDP-Prüfsumme wird weder berechnet noch überprüft

Tabelle 5-1: Eigenschaften des IP-Stacks der Universität von Queensland

Der Stack ist modular aufgebaut. Er kann als Virtuelle Komponente (VC) in ein Design eingebunden werden, ohne dass man über die Internas Bescheid wissen muss. In der Datei `globalconstants.vhd` kann die gewünschte MAC- und IP-Adresse eingetragen werden, danach kann man das eingene Design an den IP-Sender resp. Empfänger anschliessen. Im Internet ist auch eine Dokumentation dazu [13] verfügbar.

Wenn man genauer über die Internas Bescheid wissen will oder muss, so muss man sehr viel aus dem Sourcecode herauslesen. Im Anhang C sind deshalb als Hilfe auch die Zustandsdiagramme der einzelnen Layer abgebildet.

### 5.3.2 Probleme und Fehler

Tabelle 5-1 zeigt eine Auflistung der Eigenschaften und Einschränkungen des Designs. Wir überprüften nun, welche Konsequenzen die Einschränkungen auf die Funktionalität des Stacks und speziell auch auf unser Design haben könnten. Beim Testen haben wir den Stack zunächst von verschiedenen anderen Computern aus gleichzeitig mit Ping-Paketen überflutet. Dabei zeigten sich mehrere Probleme, die z.T. von den Einschränkungen her erwartet werden mussten, zum Teil aber auch durch Designfehler verursacht wurden. Einige dieser Probleme waren nicht gravierend, bei anderen drängten sich aber Anpassungen auf.

- IP-Pakete, die grösser sind als die maximale Datenfeld-Grösse eines Ethernet-Frames, müssen fragmentiert werden. Linux-Hosts scheinen aber diese Fragmente teilweise in umgekehrter Reihenfolge abzuschicken. Dies hat aufgrund der in Tabelle 5-1 genannten Einschränkung im IP-Empfänger zur Folge, dass diese Pakete vom Stack nicht empfangen werden können.
- Bei mehr als zwei gleichzeitigen Kommunikationspartnern müssen aufgrund der kleinen ARP-Tabelle (nur zwei Einträge) ständig ARP-Requests ausgesendet werden. Während dieser Zeit wird der ganze Sender blockiert (siehe auch Tabelle 5-1, ARP Einschränkungen). Das schlägt sich natürlich auch in der Performance wieder (viele verlorene Ping-Pakete).
- Der Stack muss zwingend über eine Vollduplex-Verbindung an ein anderes Gerät angeschlossen werden (Switch oder Host). Ansonsten können Kollisionen auftreten, auf die der Stack nicht reagieren kann. (Siehe Tabelle 5-1, Ethernet Einschränkungen). Schlimmer noch: Er erzeugt ständig Kollisionen, da er kein Carrier-Sense durchführt. Der schlimmste Fall tritt ein, wenn der Stack eine Kollision verursacht, indem er ein eigenes ARP-Request Paket in ein anderes, das gerade auf der Leitung ist, hineinsendet. Das ARP-Paket geht dann durch die Kollision verloren und es findet keine Retransmission statt. Dadurch wird der ARP-Request auch nicht beantwortet und der Sender ist bis zum ARP-Timeout blockiert.
- Zum effizienten Entwickeln von VHDL-Code ist es wichtig, dass der Stack simuliert werden kann. Dies war aber nicht möglich, da keine Reset-Funktionalität eingebaut war. Dadurch bleiben viele Signale in einem undefinierten Zustand, was natürlich das Debugging sehr stark erschwert.
- Der nächste Fehler trat nur gelegentlich und nicht reproduzierbar auf: Der Stack sendete manche ARP-Pakete mehrfach. Häufig doppelt, manchmal bis zu zehnfach. Der Grund dafür lag in einem Fehler in einer Zustandsmaschine. Er führte dazu, dass abhängig von einem externen Signal manchmal auf die Bestätigung für "Paket gesendet" nicht reagiert wurde und dadurch das Signal für "Sende Paket" weiterhin gesetzt blieb.
- Ein gravierendes Problem zeigte sich beim Timing des ganzen Designs. Der CRC-Generator war so aufgebaut, dass er acht Taktzyklen benötigte, um ein Byte zu verarbeiten (einen pro Bit). Dies hatte zur Folge, dass das ganze Design mit mindestens 40MHz getaktet werden musste, damit die CRC-Prüfsumme beim Senden eines Paketes zur richtigen Zeit zur Verfügung

stand. Beim Synthetisieren gaben die Tools aber eine maximale Betriebsfrequenz zwischen 24 und 29 MHz an. Es geschieht dann dasselbe, wie wenn man einen Prozessor übertaktet: Der stabile und zuverlässige Betrieb ist nicht mehr gewährleistet. Das Problem war auch den Autoren des Stacks bewusst. Sie meinen dazu in ihrer Dokumentation auf Seite 10: „..no problems have been witnessed with it running at 50MHz. ... However, sythesising and implementing with a full optimisation effort is recommended“. Ein Design aber, dass nur durch extreme Optimierung überhaupt läuft, ist meistens schlecht aufgebaut (siehe auch [6], Seite 37ff), und es war wohl reine Glückssache, dass es keine Probleme gegeben haben soll. Man muss in einem solchen Fall versuchen, eine bessere Hardware-Struktur für den benötigten Algorithmus zu finden, was wir dann auch taten.

### 5.3.3 Verbesserungen des Designs

Es gab also noch etliche Probleme beim Stack, die gelöst werden mussten. Dabei waren die drei letztgenannten ganz klar die dringendsten:

- **CRC-Generator:** Hier lag ein grundsätzlicher Designfehler vor. Das Design des CRC-Generators war für den IP-Stack überhaupt nicht geeignet. Der Generator hätte höchstens die Hälfte der Zeit brauchen dürfen, um eine Berechnung zu Ende zu führen. Glücklicherweise war er aber der einzige Block der Schaltung, der von diesem Problem betroffen war. Wir haben deshalb einen neuen CRC-Generator mit verbesserter Architektur geschrieben, der die Berechnung für ein ganzes Byte in einem Takt ausführen kann, also achtmal schneller arbeitet. Dies erlaubt, dass die Betriebsfrequenz für das ganze Design nun bis auf 5 MHz abgesenkt werden kann. Dies ist die Minimal-Frequenz, da man sonst die Ethernet-Daten, die mit einer Frequenz von 2.5 MHz anliegen, nicht mehr abtasten könnte (Nyquist-Theorem).
- **Fehler in der Zustands-Maschine:** Hierbei handelte es sich um einen klassischen Bug. Der Fehler war, dass in der Entity “ARPSnd” in den Zuständen stStoreARPRequest/Reply das Signal frameSent=1 nur dann erkannt wurde, wenn gleichzeitig das Signal complete=0 war. Das complete-Signal ist aber auch von anderen Teilen der Schaltung abhängig. Er konnte dadurch behoben werden, dass wir die Zustands-Maschine angepasst haben.
- **Simulierbarkeit:** Um dieses Problem zu lösen, mussten alle Dateien des Sourcecodes durchgegangen werden und bei allen Zustands-Automaten ein definierter Reset-Zustand eingeführt werden. Nachdem wir dies getan hatten, konnte der Stack endlich auch simuliert werden.

Da der Stack auch externe Bauteile auf dem XESS-Board benutzt, haben wir einen Testbench geschrieben, der auch das RAM und den Ethernet-PHY-Chip simuliert (siehe Kapitel 5.5).

- **CSMA/CD:** Dieses Problem lässt sich relativ einfach umgehen, indem man den Stack an einer Vollduplex-Verbindung betreibt. Wir haben deshalb darauf verzichtet, diese Funktionalität zu implementieren.

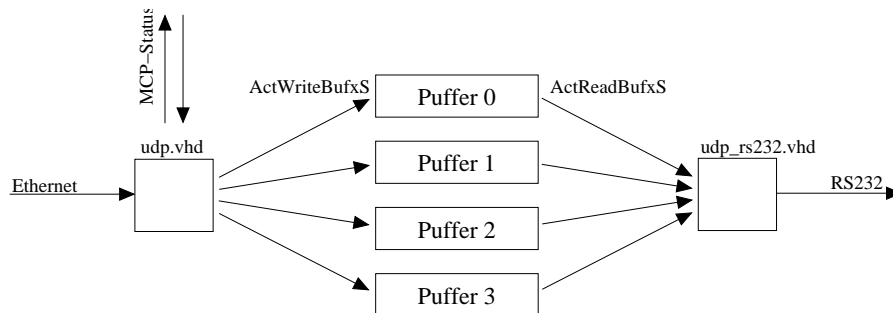


Abbildung 5-2  
Blockdiagramm des MCP-Empfängers

- **ARP-Tabelle, fragmentierte Pakete:** Diese Probleme sind wohl nur schwierig zu lösen. Hingegen lassen sie sich relativ einfach umgehen und stören deshalb in unserem Design nicht. Wenn man höchstens zwei Kommunikationspartner gleichzeitig hat, spielt die kleine ARP-Tabelle keine Rolle. Wenn man nur Pakete mit einer Grösse von maximal 1500 Bytes (max. Ethernet-Payload) verwendet, so werden sie auch nicht fragmentiert.

Nach diesen Änderungen war der Stack soweit ausgereift, dass er wirklich zuverlässig lief und wir ihn gebrauchen konnten. Das Interface von aussen her gesehen änderte sich dadurch nicht.

#### 5.3.4 Erweiterung um das MC-Protokoll

Es ging nun darum, das Vorhandene so zu erweitern, dass es via MC-Protokoll mit dem Host kommunizieren konnte. Dazu brauchte es einerseits eine Steuerung, andererseits einen Teil zur Bearbeitung der Daten.

In unserem Design wurde dieser MCP-Block in den IP-Stack integriert, weil er auch durch den darin enthaltenen RAM-Kontroller zur Koordination des Speicherzugriffs auf die verschiedenen Puffer zugreifen muss. Wir haben also den IP-Stack zu einem MCP-Stack erweitert.

Der Datenfluss durch den MCP-Block in Richtung Ethernet → RS232 ist auf Abbildung 5-2 zu erkennen. Der Block auf der Eingangsseite beachtet nur UDP-Pakete auf Port 2000. Dabei wartet er nach der Inbetriebnahme der Bridge zunächst einmal passiv auf einen MCP-Verbindungsaufbau von Seiten des Hosts. Wenn der Verbindungsaufbau erfolgreich war, werden die nun ankommenden Datenpakete in einen der vier Pufferspeicher (sie funktionieren als Ringpuffer) eingefügt. Gleichzeitig wird die mitgesendete Sequenznummer gespeichert, da sie mit dem nächsten Ack-Paket zurückgeschickt werden muss. Wenn kein Puffer mehr frei ist, wird das Paket verworfen und der Host wird es, da er keine Bestätigung dafür erhalten hat, nochmals senden. Die vier Pufferspeicher haben wir eingefügt, weil die Ethernet-Pakete viel schneller hintereinander eintreffen können als die darin enthaltenen Daten auf den RS232-Port ausgegeben werden können. Bei nur einem Puffer könnten so-

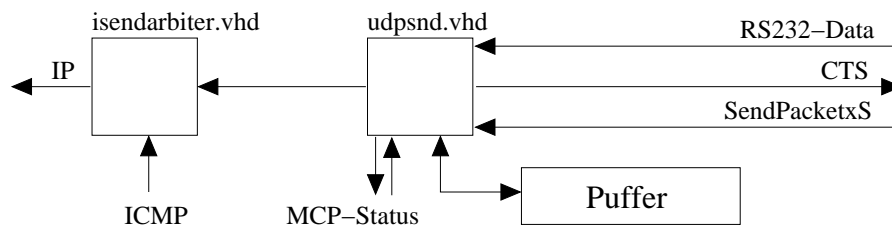


Abbildung 5-3  
Blockdiagramm des MCP-Senders

mit keine neuen IP-Pakete empfangen werden bis der Puffer wieder leer ist. Ein weiterer Vorteil ist, dass damit ein Sliding-Window-Verfahren bei der Übertragung möglich wird (siehe Kapitel 3.3.5).

Anders sieht es in Richtung RS232 → Ethernet aus. Hier musste ein Arbiter eingesetzt werden, da sowohl der MCP-Sender als auch der ICMP-Sender auf den IP-Sender zugreifen. Er koordiniert die Zugriffe der beiden Module.

Es gibt in diese Richtung nur einen Puffer (siehe Abb. 5-3). Er dient dazu, die Daten, die vom RS232-Interface kommen, zwischenspeichern und danach paketweise an den MCP-Sender weiterzuleiten. Der Ausgangsblock fügt dem Paket beim Versenden die jeweils aktuelle Seq- beziehungsweise Ack-Nummer hinzu. Der RS232-Empfänger darf während der Zeit des Versendens keine neuen Daten erhalten. Deshalb wird dem Bluetooth-Modul durch Setzen des CTS-Signals auf der seriellen Verbindung angezeigt, dass es vorläufig nicht mehr senden darf.

Der MCP-Empfänger überprüft bei jedem Paket anhand von IP-Adresse, Port, Sequenznummer und Ack-Nummer, ob es auch wirklich zur aktuellen Verbindung gehört. Ansonsten wird es verworfen. Welche Pakete akzeptiert werden, hängt auch vom aktuellen Zustand der MCP-Verbindung ab. Wenn z.B. schon eine Verbindung besteht, so werden prinzipiell alle Pakete mit gesetztem SYN-Bit ignoriert. Falls aber noch keine Verbindung besteht, werden nur SYN-Pakete angenommen.

Beim MCP-Sender hat uns leider die Zeit gefehlt, das MCP-Protokoll vollständig inklusive Retransmission zu implementieren. Deshalb wird jedes Paket nur einmal übertragen und nicht wiederholt. Die Ack- und Sequenznummern werden aber korrekt behandelt, so dass der Host nichts von dieser Einschränkung bemerkt, solange auf dem Rückweg keine Pakete verloren gehen.

Das Signal SendPacketxS, das von der Glue-Logic angesteuert wird, zeigt an, dass ein Paket mit den Daten, die in der Zwischenzeit im Puffer gesammelt worden sind, abgeschickt werden soll.

Das Zustands-Diagramm des MCP-Kontrollers ist auf Abb. 5-4 zu sehen, wobei der Config-Zustand noch nicht implementiert ist. Er wäre dafür gedacht, die Bridge am Anfang mit einer IP-Adresse zu konfigurieren (z.B. mittels DHCP).

Somit ist nun ein fertiger MCP-Stack entstanden. Er baut auf dem darunterliegenden IP-Stack auf und besitzt alle seine Funktionen. Er kann somit auch z.B. mittels

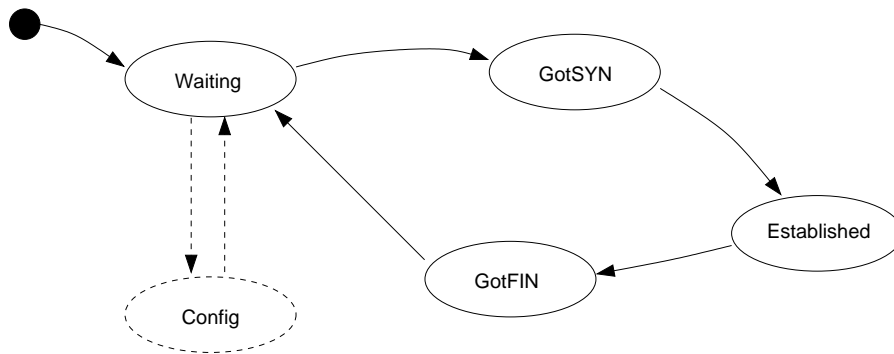


Abbildung 5-4  
Zustands-Diagramm der MCP Zustandsmaschine

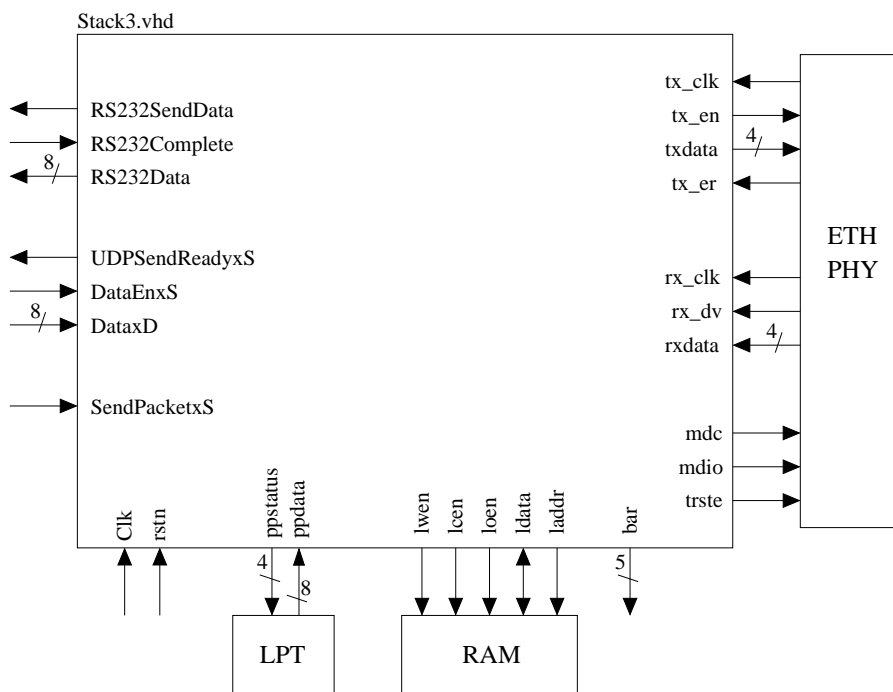


Abbildung 5-5  
Interface des IP-Stacks mit MCP

ICMP Echo-Requests "angepingt" werden. Das Interface des gesamten Blocks ist aus Abb. 5-5 ersichtlich.

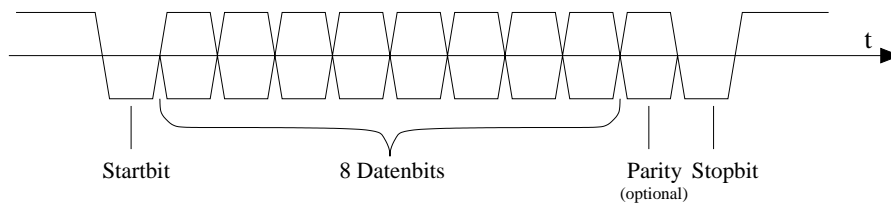


Abbildung 5-6  
Übertragung eines Bytes mit dem RS232-Protokoll

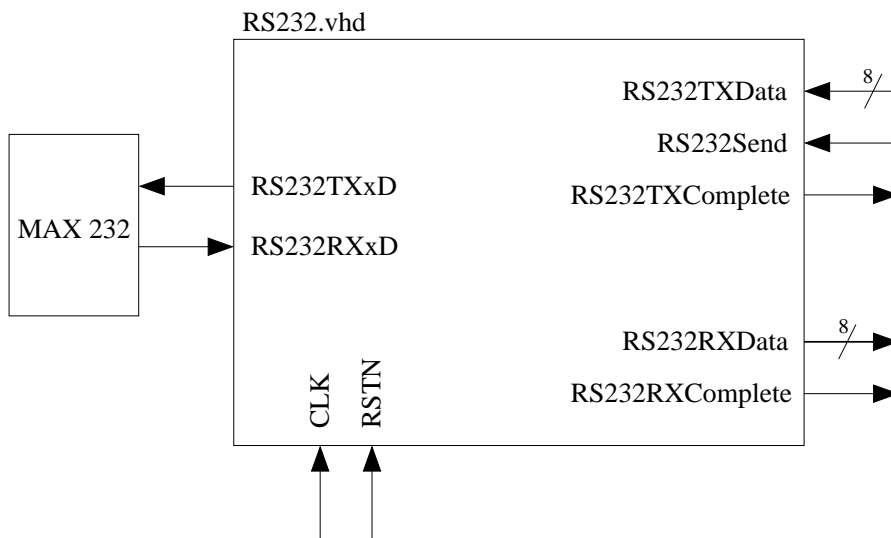


Abbildung 5-7  
Das Interface des RS232-Blocks

## 5.4 RS-232/Glue-Logic

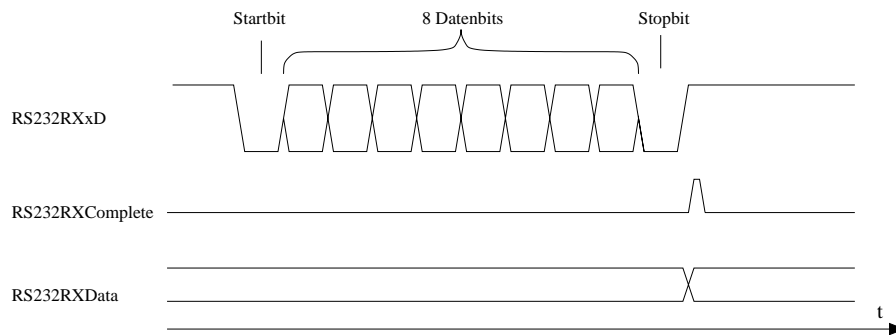
### 5.4.1 RS-232

Auf dem Board war ein MAX-232-IC für die Pegelwandlung vorhanden. So braucht es auf der RS-232-Seite noch einen Block, der das RS-232-Protokoll handhabt.

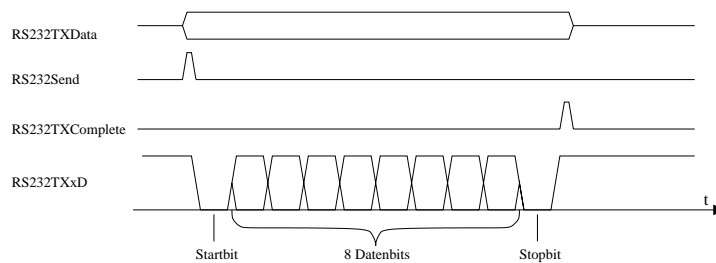
Das Protokoll sieht so aus, dass für jedes Datenwort (im Normalfall 8 Bit) ein Start-Bit, danach die Datenbits und am Schluss nach einem optionalen Paritätsbit noch ein oder zwei Stopbits gesendet werden (siehe Abb. 5-6).

Die Bitdauer ist durch die eingestellte Baudrate bestimmt und der Empfänger muss sich durch das Startbit selber richtig auf die Abtastzeitpunkte synchronisieren. Der Bittakt wird intern im FPGA durch einen Teiler erzeugt, der vom Systemtakt gespiesen wird. Der Teilungsfaktor wird folgendermassen berechnet:

$$\text{Teilungsfaktor} = \frac{\text{Systemtakt}}{\text{Baudrate}}$$



**Abbildung 5-8**  
Timing-Diagramm des RS232-Empfängers



**Abbildung 5-9**  
Timing-Diagramm des RS232-Senders

und beträgt in unserem Fall

$$\frac{16MHz}{57600bit/s} = 277.78$$

Diese Zahl muss in der Datei `rs232.vhd` bei Bedarf angepasst werden.

Sender und Empfänger bestehen je aus einer State-Maschine, die ein Byte sendet oder empfängt. Die Schnittstelle ist aus Bild 5-7 ersichtlich. Die Timing-Diagramme für RX und TX sind aus Abb. 5-8 resp. 5-9 ersichtlich.

### 5.4.2 Glue-Logic

Zwischen den beiden grossen Blöcken braucht es noch eine Glue-Logic, die den Datenaustausch koordiniert. Abbildung 5-10 zeigt ein Blockdiagramm. Sie übernimmt folgende Aufgaben:

- **Steuerung der Paketierung:** Es braucht eine Instanz, die entscheidet, wann ein neues Paket abgeschickt werden soll. Dies ist zum einen immer dann der Fall, wenn die maximale Grösse des Puffers von 1024 Bytes erreicht ist. Daneben braucht es noch einen zeitgesteuerten Auslöser, damit keine Deadlocks entstehen können. Er sollte nicht zu lange warten, da sonst Verzögerungen

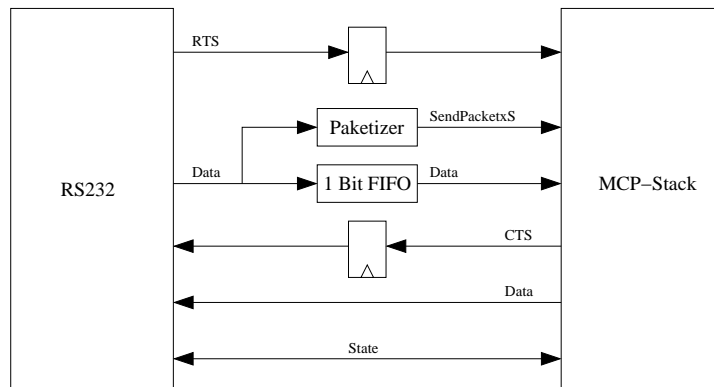


Abbildung 5-10  
Blockdiagramm der Glue-Logik

verursacht werden, aber auch nicht zu kurz, da sonst unnötig viele Pakete abgeschickt werden. Als Timeout wurde eine Zeit von 2 ms gewählt, d.h. wenn 2 ms lang keine neuen Daten beim RS232-Eingang eintreffen, wird ein Paket abgeschickt (sofern es überhaupt Daten im Puffer hat). Diese Zeitdauer hat sich bei unseren Tests bewährt.

Als weiterer Auslöser für das Versenden eines Paketes dient ausserdem das Eintreffen eines neuen Datenpaketes vom Host her. So kann sofort ein ACK-Paket zurückgeschickt werden, was den Vorteil einer kürzeren Round-Trip-Zeit hat.

- **Steuerung der RS232-Flusskontrolle:** Mittels der beiden Signale RTS und CTS kann eine Flusskontrolle zwischen der E2B-Bridge und dem Bluetooth-Modul realisiert werden. Das Modul zeigt mit RTS an, dass es momentan keine Daten mehr über die serielle Schnittstelle empfangen kann (z.B. weil sein Eingangspuffer voll ist). Die E2B-Bridge ihrerseits benutzt das CTS-Signal, um dem Modul anzuzeigen, dass es keine weiteren Daten schicken soll. Die Glue-Logic kontrolliert diese beiden Signale.

RTS bewirkt, dass der Teil des MCP-Layers, der die Daten aus einem der vier Puffer ausliest, seine Arbeit unterbricht. CTS wird immer dann gesetzt, wenn der Stack damit beschäftigt ist, ein Paket zu verschicken. So bleibt der Puffer im alten Zustand.

Es kann sein, dass das Modul gerade ein Byte am Senden ist, wenn die Bridge das CTS-Signal setzt. Das Byte wird in diesem Fall trotzdem zu Ende übertragen, erst danach erfolgt die Pause. Weil die Bridge aber schon am Abschicken des Paketes ist, wenn dieses letzte Byte eintrifft, musste in der Glue-Logik noch ein 1-Bit FIFO integriert werden, das dieses Byte zwischenspeichert.

- **Synchronisation der RTS/CTS-Signale:** Die beiden Signale müssen mit einem Register auf den internen Takt im FPGA synchronisiert werden, damit die Setup-Zeiten stets eingehalten werden und keine Glitches an den FPGA-Ausgängen erzeugt werden.

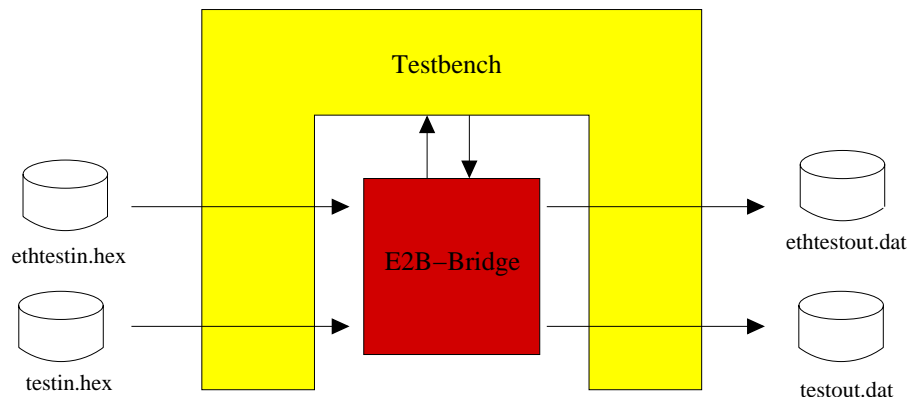


Abbildung 5-11  
Datenfluss-Diagramm des Testbenchs

## 5.5 Simulation

Das fertige Design belegt auf dem Virtex800 24% der Fläche. Der Stack alleine benötigt schon 21%. Dies hat zur Folge, dass die Synthese und Implementierung ca. 10 Minuten benötigen, so dass es nicht mehr effizient ist, jede kleine Änderung auf dem FPGA auszutesten.

Eine Simulation des Designs bietet hier viele Vorteile: Man kann innert einer Minute testen, ob das Design funktioniert. Falls etwas nicht stimmt, kann in *Modelsim* der zeitliche Verlauf jedes einzelnen Signals beobachtet und sogar beeinflusst werden. Dies ist auf dem FPGA natürlich nicht möglich und eine enorme Hilfe beim Debugging.

Wir haben einen Testbench geschrieben, um unser ganzes Design der Bridge zu simulieren. Der Testbench muss dazu alle externen Komponenten nachbilden. Er generiert alle Taktsignale und simuliert die RAM-Bausteine und die Ethernet- und RS232-PHY-Chips. Die Simulation der beiden letzteren wird aus zwei Dateien gesteuert, aus denen Testdaten (Stimuli) ausgelesen werden. Die beiden Dateien heißen *ethtestin.dat* (für Ethernet-Pakete) und *testin.dat* (RS232-Daten).

Die Ausgaben (Responses) werden in zwei Dateien abgespeichert, sie heißen entsprechend *ethtestout.dat* und *testout.dat*. Alle Dateien für die Simulation sind in den Unterverzeichnissen `Diplomarbeit/E2B-Bridge/Hardware/modelsim` und `Diplomarbeit/E2B-Bridge/Hardware/Testvektoren` zu finden.

Nach einem Simulationsdurchlauf kann in den beiden Dateien überprüft werden, ob auch wirklich das Erwartete ausgeführt wurde. Diese Anordnung erlaubt es, ganze Abläufe mit Verbindungsaufbau, Datenübertragung in beide Richtungen und darauf folgendem Verbindungsabbau durchzuspielen und somit eine funktionale Verifikation des Designs durchzuführen.

Simulation war für uns eine gute und schnelle Methode, um kleine Änderungen am Design auf ihre Richtigkeit hin zu überprüfen und das Verhalten in gewissen Spezialfällen zu überprüfen, was im laufenden Betrieb fast unmöglich ist. Der Nachteil

war die umständliche Erstellung der Stimuli-Files für grosse Simulationen und das Überprüfen der erzeugten Responses. Deshalb haben wir für Praxistests das ganze Design zwischendurch immer wieder synthetisiert und auf dem FPGA laufen lassen.

# 6

## *Resultate und Fazit*

In Kapitel 6.1 wird erläutert, wie und mit welchem Resultat wir unsere Hardware und Software getestet haben. In Kapitel 6.2 wird dann auf die Erfahrungen eingegangen, die wir während unserer Arbeit gesammelt haben. In Kapitel 6.3 werden die Schlussfolgerungen gezogen.

### *6.1 Test-Szenarien*

Die Funktionstüchtigkeit sowohl der E2B-Bridge als auch des gesamten Konzepts wurde mit zwei Szenarien überprüft.

#### *6.1.1 Test des MC-Protokolls*

Wir haben die Bridge mittels “Netty” an den Bluez-Stack angeschlossen. Daraufhin haben wir “l2ping” und “l2test” einige Male über Nacht laufen lassen. Dabei kam es zunächst gelegentlichen zum Verlust von einem einzelnen Byte (so etwa alle 20 Minuten mit l2test). Beim Untersuchen dieses Problems haben wir festgestellt, dass auf dem Host durch diverse Logging-Funktionen so viel Last erzeugt wurde, dass er nicht mehr nachgekommen ist mit dem Verarbeiten der eintreffenden Daten. Dadurch gingen einzelne Bytes verloren.

Nach Verringerung des Debug-Levels und dem Schliessen einiger anderer Programme hat die Bridge dann tadellos funktioniert. Dabei scheint vor allem die gleichzeitige Benutzung des CDROM-Laufwerks ein Problem gewesen zu sein.

Sehr wichtig ist, dass bei den angeschlossenen Modulen die Flusskontrolle per RTS/CTS funktioniert. Die Module, die uns zur Verfügung standen, hatten teilweise einen Fehler, so dass dies nicht der Fall war. Dies ist deshalb wichtig, da die Bridge ansonsten dem Modul nicht mitteilen kann, dass ihr Puffer voll ist. Das Modul würde in diesem Fall einfach weiter senden, und die Daten gingen verloren.

## 6.1.2 Demonstrations-Applikation

Um das ganze Konzept zu überprüfen haben wir eine Demonstrations-Applikation entworfen. Diese misst die Empfangsqualität aller Bluetooth-Verbindungen einer E2B-Bridge. Diese Messungen stellen die Basis für Handover-Entscheidungen und Positionsbestimmungen dar.

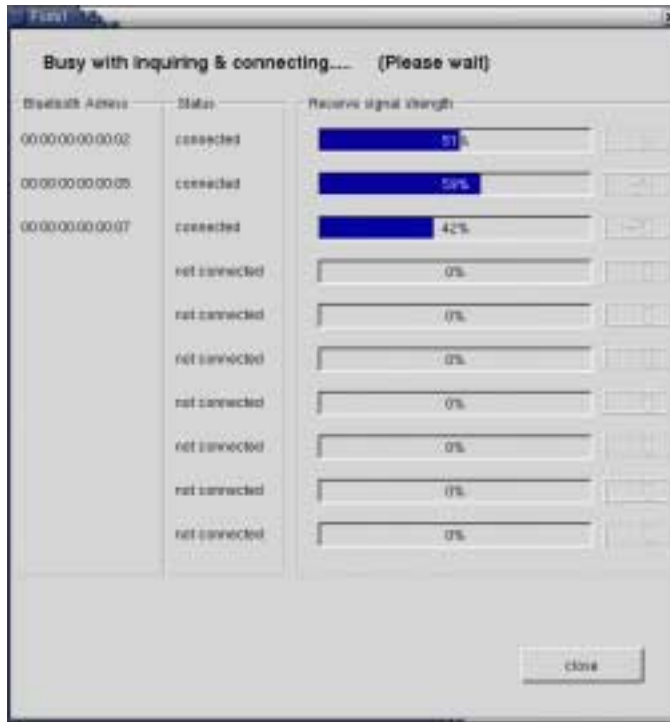


Abbildung 6-1  
Screenshot der  
Demonstrations-  
Applikation

Wie in Abbildung 6-1 gesehen werden kann, funktioniert die Applikation einwandfrei.

## 6.2 Erfahrungen und Einschränkungen

### 6.2.1 E2B-Bridge

Die E2B-Bridge ist soweit funktionsfähig, sie könnte noch an ein paar Stellen verbessert und vervollständigt werden. Das meiste davon sind Kleinigkeiten, die in relativ kurzer Zeit noch ergänzt werden könnten. Folgende Ergänzungen wären noch möglich:

- **Rückweg mit MCP:** Das vollständige MC-Protokoll mit Fehlerkorrektur und Retransmission wird momentan nur in Richtung Host → Bridge benutzt. Auf dem Rückweg wird das Paket nur einmal gesendet. Hier müsste man noch einen Mechanismus für die Zwischenspeicherung von mehreren Paketen und

für Retransmissions nach einem Timeout einbauen. Dies hätte auch den Vorteil, dass während dem Verschicken eines Paketes Daten vom Modul zwischengespeichert werden könnten. Uns hat leider die Zeit zu diesem Schritt nicht mehr gereicht.

- **Konfiguration:** Die Bridge arbeitet momentan mit einer statischen IP- und MAC-Adresse. Auch die RS232-Bitrate ist konstant. Änderungen an diesen Werten können nur direkt im Sourcecode vorgenommen werden.

Dies ist etwas unflexibel und deshalb könnte man in einer zukünftigen Version der E2B-Bridge hier einen Mechanismus einbauen, damit sie z.B. über das Netz konfiguriert werden könnte. Es wäre z.B. denkbar, dass für das Finden einer gültigen IP-Adresse das DHCP-Protokoll zum Einsatz kommen könnte.

Für die Konfiguration der RS232-Schnittstelle müsste die Bridge von der Software "Netty" die Information erhalten, welche Bitrate gewünscht ist. Man könnte zum Beispiel ein Bit im MCP-Header benutzen, um das Paket als Konfigurations-Paket zu markieren. Die Daten enthalten in diesem Fall die gewünschte Bitrate, die die E2B-Bridge dann übernimmt.

- **Standard-Gateway:** Die E2B-Bridge kann noch nicht mit Standard-Gateways zusammenarbeiten, da sie immer direkt die IP-Adresse des Kommunikationspartners aufzulösen versucht. Man müsste noch eine Definition eines Standard-Gateways und einer Netzmaske hinzufügen. ARP würde, falls die Destination eines Paketes nicht im eigenen Netz liegt, die MAC-Adresse des Standard-Gateways suchen.
- **CSMA/CD:** Die Bridge darf momentan nur via Switches oder direkt an den Host angeschlossen werden, da der Ethernet-Sender kein Carrier-Sense durchführt. Damit sie auch an ein Halbduplex-Netzwerk angeschlossen werden kann, müsste der MAC-Layer des IP-Stacks um die CSMA/CD-Funktionalität erweitert werden.

### 6.2.2 Netty

Die Software funktioniert soweit einwandfrei, auch wenn mehrere E2B-Bridges angeschlossen sind. Nur der Verbindungsabbau ist noch nicht vollständig getestet. So muss folgender Punkt noch implementiert werden:

- **Fehlermanagement-Timeouts:** Ein Keepalive-Timeout und ein Timeout beim Verbindungsabbau gemäss Kapitel 3.3.6 würde die Fehlertoleranz stark vergrössern.

### 6.2.3 Bluetooth-Module

Die Bluetooth-Module sind in der Funktionalität noch stark eingeschränkt (siehe auch Kapitel 2.2). So sind gewisse Szenarien unserer Vision noch nicht implementierbar, da gewisse Funktionen wie z.B. Point-to-Multipoint oder Read\_RSSI noch nicht von allen Modulen unterstützt wird.

### *6.2.4 Bluez*

Der verwendete Bluetooth-Stack ist bereits sehr stabil. Nur mit dem Verbindungsabbau scheinen noch gewisse Probleme zu bestehen. So kann die xhop-Funktionalität [7] noch nicht mit zufriedenstellender Geschwindigkeit ausgeführt werden. Doch die Entwickler sind sich der Problematik bewusst und das Problem soll in den zukünftigen Versionen behoben werden.

### *6.3 Fazit*

In unserer Arbeit haben wir gezeigt, dass eine Bluetooth-Ethernet-Bridge auf FPGA-Basis möglich ist und eine erste Version dieser Bridge mit dazugehöriger Software auf Host-Seite aufgebaut. Sie ist funktionsfähig und es ist möglich, an einem Host mehrere Bridges parallel zu betreiben.

Durch entsprechende zusätzliche Software und eine Vervollständigung der Bridge-Funktionalität ist es möglich, die in Kapitel 1 vorgestellte Vision mit verteilten Bluetooth-Access-Points zu realisieren.

# 7

## *Dynamische Rekonfiguration des FPGAs*

In den letzten drei Wochen unserer Arbeit eröffnete sich die Möglichkeit, uns mit einem aktuellen Forschungsgebiet zu befassen. Dabei entschieden wir uns für das am TIK intensiv untersuchte Gebiet des *Reconfigurable Computing*. Es ging hier mehr um grundsätzliche Untersuchungen ohne direkten Zusammenhang mit der E2B-Bridge. In diesem Kapitel zeigen wir, was wir untersucht haben und was dabei herausgekommen ist.

Im ersten Kapitel erklären wir den internen Aufbau der Xilinx Virtex FPGAs und wie diese dynamisch und partiell rekonfiguriert werden können. In den folgenden Kapiteln gehen wir auf unsere Arbeit mit *JBits* ein und zeigen, was wir untersucht haben. Kapitel 7.6 fasst dann unsere Resultate zusammen.

### *7.1 Die Xilinx-Virtex Architektur und Rekonfiguration*

Die Xilinx Virtex FPGAs bestehen je nach Typ aus verschieden vielen *CLBs* (Configurable Logic Blocks), die in einem zweidimensionalen Array angeordnet sind. Die CLBs sind untereinander durch ein Gitter von konfigurierbaren Routen verbunden. Die Logik in den CLBs und die Routen können durch einen sogenannten Bitstream konfiguriert werden. Am Rand des Chips sind die IOBs (Input/Output Blocks) angeordnet. Diese sind mit den Pins des Chipgehäuses verbunden und dienen so der Verbindung mit der Aussenwelt. Die *Xilinx Architecture Description* [17] beschreibt den inneren Aufbau dieser Blöcke.

Die Virtex-Familie von Xilinx gehört zu den *SRAM-basierten FPGAs*. Das bedeutet, dass die Konfiguration intern in SRAM-Speicherzellen gespeichert wird. Der Nachteil davon ist natürlich, dass nach jedem Unterbruch der Versorgungsspannung die ganze Konfiguration neu geladen werden muss. Xilinx hat aber auch den Vorteil der flexiblen Änderbarkeit der SRAM-Zellen nutzbar gemacht: Bei den

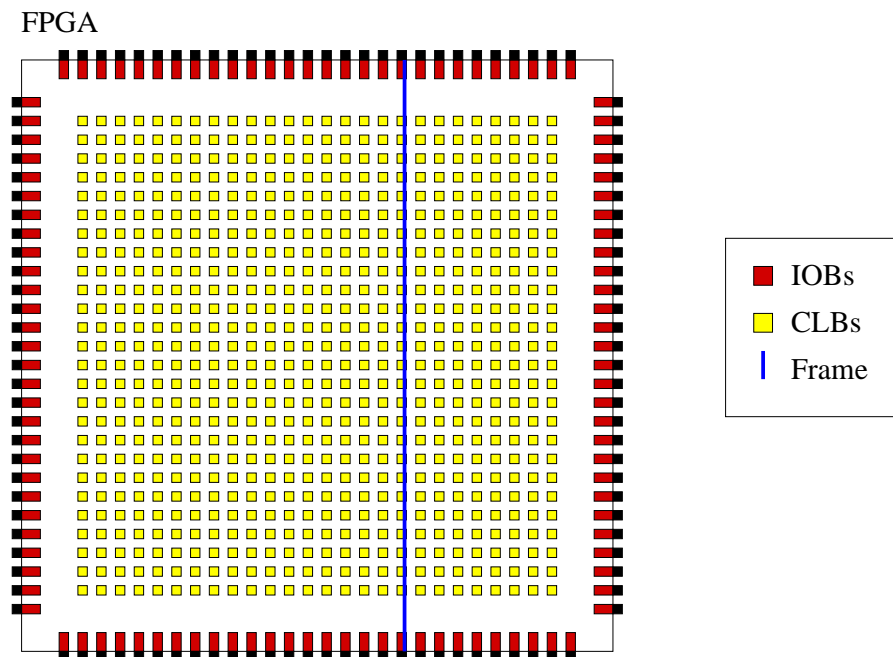


Abbildung 7-1  
Schematische Darstellung des Aufbaus eines Virtex FPGAs

Virtex-Bausteinen ist der Controller, der die Konfiguration in den FPGA hineinlädt, so aufgebaut, dass die Konfigurationsdaten ähnlich wie bei einem RAM-Baustein adressiert werden können. Man kann damit also nicht nur eine volle Konfiguration hineinschreiben, sondern es können auch nur *partielle Konfigurationen* heruntergeladen werden, die nur einen Teil der internen Funktionalität ändern. Es ist ebenfalls möglich, Konfigurationsdaten wieder aus dem FPGA auszulesen (dies ist für Debugging-Zwecke sehr nützlich, weil der aktuelle Zustand der internen Register im ausgelesenen Konfigurationsstrom enthalten ist). Dieser Mechanismus ermöglicht es sogar, dass man gezielt bestimmte Bereiche des FPGAs dynamisch (also auch während der Laufzeit) rekonfigurieren kann.

Leider kann man nicht jedes einzelne Bit direkt ansprechen, sondern man muss immer jeweils ein ganzes sogenanntes *Frame* auslesen oder neu hineinschreiben. Ein Frame besteht aus einem Teil einer CLB-Spalte. Darin sind sowohl Teile der Konfiguration der IOBs unten und oben, als auch solche der entsprechenden CLB-Spalte (Logik, Initialisierungswerte der Register, Routing) enthalten (siehe Abbildung 7-1).

Mittels dynamischer Rekonfiguration sind viele interessante Anwendungen möglich. Man könnte sich z.B. ein *System on a Chip* vorstellen, das seine Funktionalität bei Bedarf *on demand* an die Erfordernisse anpassen kann. Dieses Ziel wird unter anderem vom Zippy-Projekt [20] verfolgt.

Xilinx ist einer der führenden FPGA-Hersteller auf dem Gebiet der rekonfigurierbaren Logik. In ihrer *FAQ* zu partieller Rekonfiguration [18] steht, dass sie in Zukunft viele kommerzielle Anwendungen von dynamischer Rekonfiguration erwar-

ten. Noch ist es aber weitgehend ein Forschungsgebiet.

Es stellt sich noch die Frage nach der Generierung der partiellen Konfigurationsdaten: Die Standard-Tools liefern am Ende des Design-Flows eine Datei, die eine volle Konfiguration enthält. Sie eignen sich aber nicht dazu, partielle Bitstreams zu erzeugen. Dazu braucht man zusätzliche Tools, wie z.B. das im nächsten Kapitel beschriebene *JBits*-Paket von Xilinx.

## 7.2 *JBits*

### 7.2.1 *Das JBits-Paket*

*JBits* ist eine Sammlung von Java-Packages von Xilinx, die eine Bearbeitung eines Konfigurations-Bitstreams ermöglichen. Ausgangspunkt bildet entweder eine volle Konfiguration, die mit einem Standard-Tool erzeugt wurde, oder eine leere Konfiguration. Diese kann man nun auf alle erdenklichen Arten bearbeiten: Ergänzen, Teile ausschneiden und kopieren, Routen ändern u.s.w. Ein Tutorial und weitere Dokumente findet man in der *JBits-Dokumentation* [15].

Die internen Strukturen des FPGAs werden durch verschiedene miteinander verbundenene *Java-Objekten* abstrahiert. Zuoberst in der Hierarchie ist das *JBits-Objekt*. Es besitzt Methoden zum Einlesen eines Bitstreams, zum Abfragen, Setzen und Löschen jedes einzelnen Konfigurationsbits, zum Speichern eines Bitstreams (voll oder partiell) u.s.w. Man kann diese Methoden direkt benutzen oder aber auch indirekt durch Anwendung eines der vielen weiteren Objekte, die darauf aufbauen und die Methoden des *JBits-Objekts* benutzen. Mit ihnen kann man die Logik in einem CLB manipulieren und Routen verfolgen, löschen und automatisch oder manuell neu setzen.

Prinzipiell kann jedes einzelne Bit des Bitstreams gezielt bearbeitet werden. Ein Problem für den Anwender ist aber, dass grosse Teile von *JBits* zur Zeit nur schlecht oder gar nicht dokumentiert sind. Laut Xilinx ist dies zum einen darin begründet, dass *JBits* das Reverse-Engineering von Designs aus Xilinx-FPGAs erleichtert. Genau dies möchte aber Xilinx möglichst vermeiden. Zum anderen will Xilinx gewisse besonders "gefährliche" Funktionen nicht offenlegen, mit denen man die FPGAs bei Experimenten mit *JBits* leicht zerstören könnte. (Man kann Bitstreams erzeugen, die im FPGA interne Kurzschlüsse verursachen.)

### 7.2.2 *Wie wird JBits angewendet?*

*JBits* ist, wie schon gesagt, eine Sammlung von Java-Packages. Deshalb braucht man zur Benutzung auch ein bereits installiertes JDK. Für die von uns benutzte *JBits*-Version 2.8 empfiehlt Xilinx das JDK 1.2.2 von Sun. Dieses kann von der Java-Homepage [16] kostenlos heruntergeladen werden. Es gibt Versionen für alle gängigen Betriebssysteme.

Der Anwender kann nun Java-Programme schreiben, die auf die Klassen in den Packages zugreifen. Ein Programmrumpf sieht folgendermassen aus:

```
import java.io.*;
import com.xilinx.JBits.Virtex.*;

public class Test {
    public static void main(String args[]) {
        String inFileName = "bitstream.bit";
        String outFileName = "bitstreamout.bit";
        int deviceType = 0;

        JBits jbits = null;

        /* Get the device type */
        deviceType = Devices.getDeviceType("XCV800");

        /*Creating a JBits Object for the Virtex device*/
        jbits = new JBits(deviceType);

        /*Reading in a Xilinx bitfile*/
        try {
            jbits.read("bitstream.bit");
        } catch (FileNotFoundException fe) {
            System.out.println(fe);
        } catch (IOException io) {
            System.out.println(io);
        } catch (ConfigurationException ce) {
            System.out.println(ce);
        }

        // Enable CRC. Nothing works if we don't do that.
        jbits.enableCrc(true);

        // functionality here

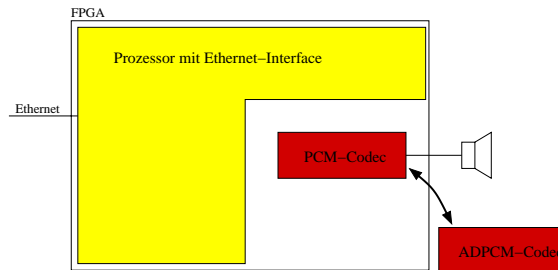
        /*Writing the modified bitstream to the file*/
        try {
            jbits.write(outFileName);
        } catch (IOException io) {
            System.out.println(io);
        }
    }
}
```

In der Mitte des Programm-Rumpfs kann nun die gewünschte Funktionalität eingefügt werden. Sehr nützlich sind Funktionen für das Routen, das Unrouten sowie das Verfolgen von Verbindungen. Im Tutorial, das in der *JBits-Dokumentation* [15] enthalten ist, werden diese und viele weitere Möglichkeiten mit Beispielen erklärt.

### 7.3 Anwendung: Multitasking in einem FPGA

Ein Block, der auf einem FPGA eine bestimmte Aufgabe ausführt, kann als Task bezeichnet werden. Eine Anwendung von *JBits* ist, dass man damit mehrere solcher *Tasks* in einem FPGA parallel laufen lassen kann. Man könnte z.B. von einer Grundkonfiguration ausgehen und danach, je nach Bedarf, mittels *JBits* und partieller Rekonfiguration neu benötigte Tasks dynamisch dazuladen und nicht mehr benötigte wieder entfernen. Die Bezeichnung *Task* kommt von der Analogie zu einem Task auf einem Multitasking-fähigen Prozessor. Die Instanz, welche entschei-

det, welche Tasks zu welchem Zeitpunkt auf dem FPGA laufen dürfen, nennt man ebenfalls analog *Hardware Operating System*.



*Abbildung 7-2  
Je nach Bedarf  
wird der richtige  
Audio-Codec  
dynamisch  
nachgeladen.*

Den Platz, den ein Task auf dem FPGA benutzt, kann als *Footprint* bezeichnet werden. Verschiedene Tasks, die gleichzeitig auf demselben FPGA laufen sollen, müssen nun so platziert bzw. verschoben und verändert werden, dass sich ihre Footprints nicht überschneiden. Falls dies nicht möglich ist und wir ein nicht präemptives FPGA-OS haben, muss der eine Task warten bis der andere beendet ist, damit er den nun frei werdenden Platz einnehmen kann.

Ein Beispiel hierfür ist [31] (siehe Abb 7-2). Hier wurde ein Design entworfen, das einen Prozessor-Core mit Ethernet-Schnittstelle enthält. Dieser kann nun Audio-Signale via Ethernet empfangen und der FPGA lädt je nach Format der Audio-Daten automatisch einen PCM- oder einen ADPCM-Codec nach. Das decodierte Signal wird danach auf einem Lautsprecher ausgegeben.

## 7.4 Task-Transformationen

Damit man den FPGA möglichst gut ausnützen kann, wäre es wünschenswert, dass man die verschiedenen Tasks flexibel an verschiedenen Stellen im FPGA einsetzen könnte.

Zusammen mit entsprechenden Placement-Algorithmen ermöglicht dies eine effizientere Auslastung des FPGAs. Eine genauere Analyse mit einem Vergleich mehrerer Strategien ist in [19] zu finden.

Wir haben folgende Task-Transformationen untersucht:

### 7.4.1 Relokation

Die einfachste Variante ist die Verschiebung und Umplatzierung eines Tasks. Man bezeichnet sie als *Relokation*. Sie ist immer dann möglich, wenn keine örtlich vorbestimmten Strukturen auf dem FPGA benutzt werden. Wenn man kein präemptives Hardware-OS benutzt, muss ansonsten gewartet werden, bis die gewünschten Ressourcen wieder frei werden.

### 7.4.2 Footprint-Transformationen

Wenn man dabei den Footprint eines Tasks durch bestimmte Operationen verändert, spricht man von einer Footprint-Transformation.

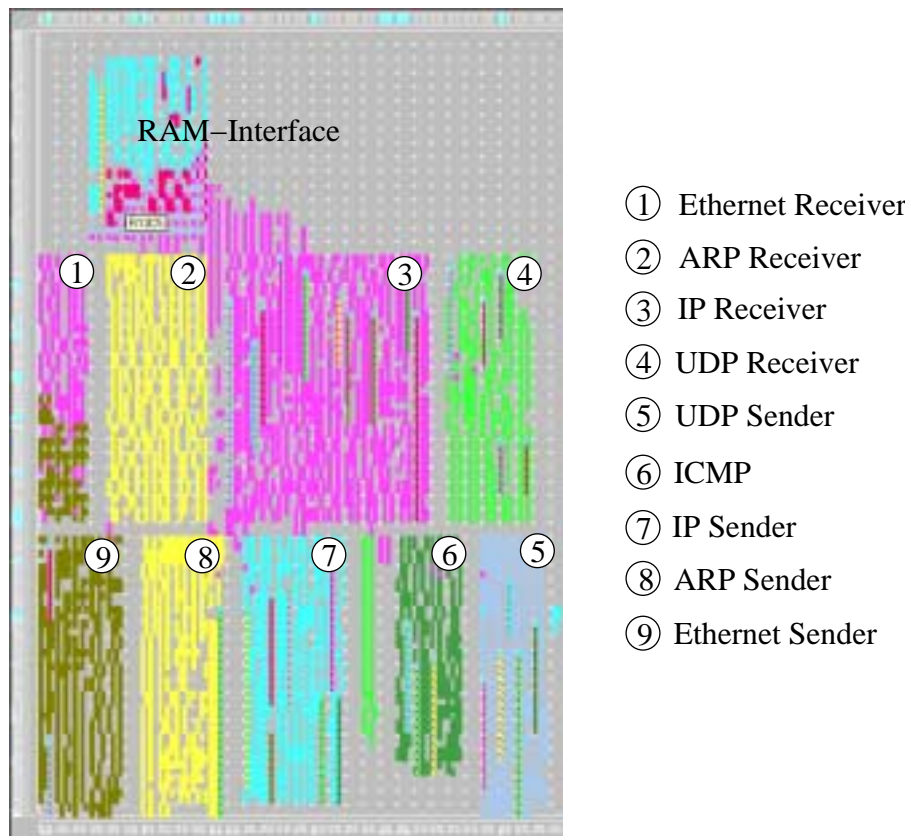


Abbildung 7-3  
Unterteilung des IP-Stacks in Subtasks

#### 7.4.2.1 Footprint-Transformation auf Subtask-Ebene

Voraussetzung für eine Footprint-Transformation auf Subtask-Ebene ist, dass der Task bereits in *Subtasks* zerlegt worden ist. Das Ziel dabei ist, die Subtasks so zu wählen, dass durch die Unterteilung möglichst wenige Verbindungsnetze durchschnitten werden. Deshalb entsprechen sie meist den Entities im VHDL-Code. Der Footprint der Subtasks muss schon im Standard-Designflow mittels Area-Constraints festgelegt werden. Bild 7-3 zeigt, wie der IP-Stack aus der E2BB in verschiedene Subtasks unterteilt werden kann.

Unter einer *Subtask-Transformation* versteht man nun die Änderung des Footprints, indem Subtasks je nach Bedarf umgeordnet werden, damit der gesamte Task eine günstigere Form einnimmt. So kann man die Platzierung auf dem FPGA weiter optimieren. Ziel unserer Arbeit war es, zu zeigen, inwiefern Subtask-Transformation im Zusammenhang mit JBits möglich ist. Als Testobjekt sollte der IP-Stack unserer E2B-Bridge dienen.

### 7.4.2.2 Footprint-Transformation auf CLB-Level

Wir haben auch die Durchführbarkeit einer weiteren Task-Transformation untersucht: Derjenigen auf CLB-Level. Hierbei versucht man, durch Verschieben von ein paar wenigen CLBs eines Tasks seinen Footprint in eine günstigere Form zu bringen (siehe Abb. 7-4). Es ist jedoch anzunehmen, dass der (Routing-)Aufwand schnell ansteigt, wenn man viele CLBs verschieben will. Deshalb wird diese Methode wahrscheinlich nur da Sinn machen, wo schon ein Verschieben von wenigen CLBs reicht, um den Task platzieren zu können.

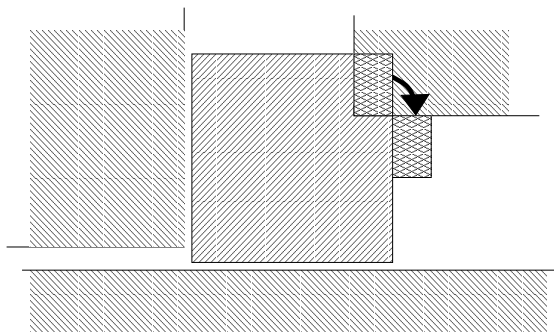


Abbildung 7-4  
Beispiel für eine  
CLB-Transformation

## 7.5 Versuche mit JBits

Da das Design der E2B-Bridge relativ gross und komplex ist, haben wir für unsere Versuche mit *JBits* zunächst ein kleineres Design ausgewählt, das übersichtlicher ist. Wir realisierten im Standard-Designflow zwei separate Designs mit je einem Task. Das eine enthielt einen 34-Bit Zähler, das andere einen Komparator mit vier Bit breitem Eingang und sieben Bit breitem Ausgang. Mit Hilfe von Area-Constraints wurden sie auf dem FPGA so platziert, dass sie sich beim zusammenkopieren nicht überschneiden würden. Es konnte jeweils eines der beiden Designs auf den FPGA geladen werden, und mittels der Siebensegment-Anzeigen konnten wir überprüfen, dass sie korrekt funktionierten.

### 7.5.1 Zusammensetzen zweier Designs

Als erstes haben wir nun ein Programm geschrieben, das beide Task zusammen auf einen FPGA kopiert. Das neu entstehende Design soll dann so aufgebaut sein, dass der Komparator-Task die linke Siebensegment-Anzeige benutzt, der Zähler-Task die rechte. Beide sollen gleichzeitig unabhängig voneinander laufen (siehe Abb. 7-5). Das *JBits-Programm* muss dazu beide Designs laden, danach den Task aus dem einen heraus- und in das andere hineinkopieren. Wir haben dazu das Programm *JBitsCopy* von Herbert Walder als Vorlage benutzt. Es kopiert die gewünschten CLBs mitsamt Logik und dem in ihnen enthaltenen Routing. Weil auch das Routing mitkopiert wird, darf man nicht vergessen, im alten Design schon vor dem Kopieren alle Routen zu entfernen, die den Task mit der Aussenwelt verbinden, ansonsten kann es sein, dass Reste dieser Leitungen mitkopiert werden. Dies könnte

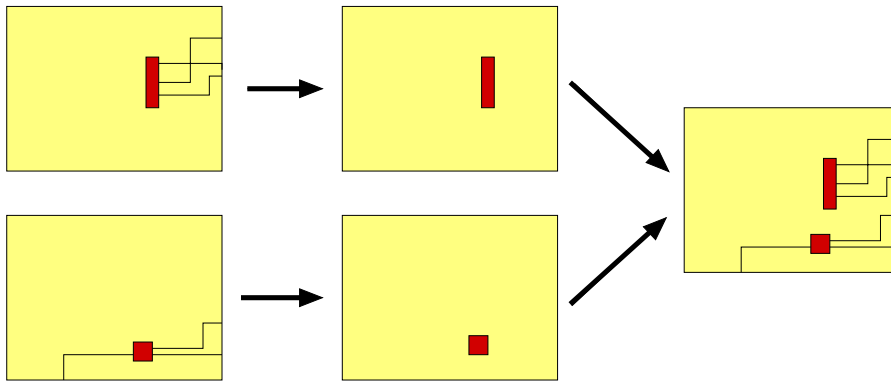


Abbildung 7-5  
Funktion des Programms Test1.java

bewirken, dass danach im neuen Design Teile von Routen “herumliegen”, was evtl. Fehler hervorgerufen könnte, wie z.B. Kurzschlüsse (siehe Abb. 7-6).

Der zweite Task wird aber nach dem Kopieren alleine noch nicht funktionieren, weil seine Verbindungen zur Aussenwelt zuerst wieder hergestellt werden müssen. Dazu kann der Autorouter *JRouter* von *JBits* benutzt werden. Er erwartet als Parameter zwei sogenannte *Pins-Objekte* (Ein- oder Ausgänge der CLBs oder IOBs). Er sucht nun automatisch eine Route, mit der er die beiden verbinden kann.

Mehrere Pins (eine Source und mehrere Sinks) können zu einem *NetPins-Objekt* zusammengefasst werden. Ausserdem gibt es ein Objekt namens *NetPinsList*, in dem wiederum verschiedene *NetPins-Objekte* zusammengefasst werden können. In einer solchen *NetPinsList* merkt sich das Programm beim Entfernen der Verbindungen die betroffenen Pins, damit es nachher noch weiss, wohin die Routen im Ausgabe-Bitstream wieder verbunden werden müssen.

Beim Entfernen gehen wir so vor, dass wir von den IOBs, die durch das Design des Prototyp-Boards vorgegeben sind, ausgehen und suchen, wo die Verbindungen

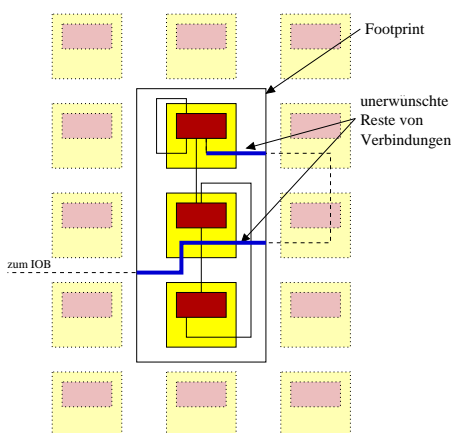


Abbildung 7-6  
Falls die Routen nicht zuerst entfernt werden, bleiben beim Kopieren unerwünschte Reste zurück.

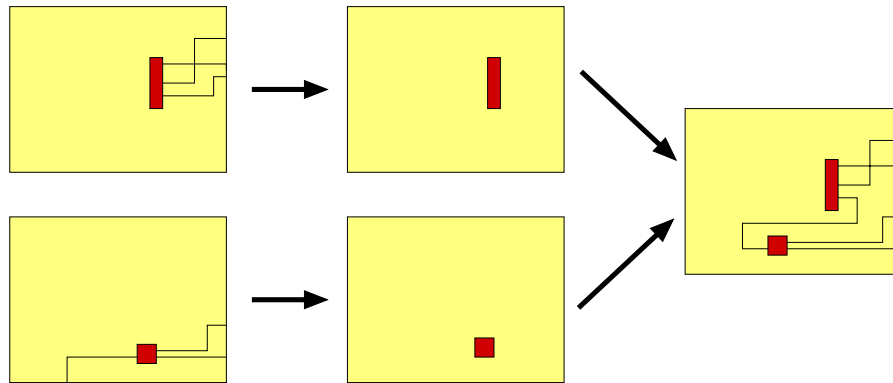


Abbildung 7-7  
Funktion des Programms Test2.java

hinführen. Diese Pins werden dann im NetPinsList-Objekt abgespeichert. So findet man jedesmal die richtigen Pins innerhalb des Tasks, da sich diese bei einer erneuten Implementation mit den Standardtools ändern können.

Ein wichtiger Punkt ist zudem der Takt: Mit dem Bitstream wird er im FPGA nur an den Stellen eingeschaltet, wo er auch tatsächlich benötigt wird. Man muss deshalb darauf achten, dass dort, wo ein neuer Task hinkopiert wird, das Clocknetz auch eingeschaltet wird. Ansonsten würden sequentielle Schaltungen, die an eine freie Stelle kopiert werden, keinen Takt erhalten und dadurch nicht funktionieren. Bei unseren Versuchen haben wir der Einfachheit halber jeweils auf dem ganzen Chip das Clocknetz eingeschaltet, damit auch sicher überall auf dem FPGA die Taktversorgung gewährleistet ist. Dies hat zwar einen höheren Stromverbrauch zur Folge, hat aber auf den Clock ansonsten keine Auswirkungen. Das Clocknetz genügend gross dimensioniert, um den ganzen Chip versorgen zu können (siehe [18]).

Entstanden ist das Programm Test1.java. Der Aufruf sieht folgendermassen aus:

```
java myself.test.Test1 <input1.bit> <input2.bit> <outfile.bit>
```

Es kopiert die beiden Tasks aus *infile1.bit* und *infile2.bit* in den neuen Bitstream *outfile.bit*. Zum Test haben wir diesen auf den FPGA heruntergeladen und alles funktionierte wunschgemäss.

### 7.5.2 Verbinden und Verschieben zweier Tasks

Darauf wurde ein zweites Programm erarbeitet, dass die beiden Tasks untereinander verbinden soll. Es nimmt als Ausgangspunkt wieder die beiden kleinen Designs. Zunächst werden alle Routen entfernt. Danach wird der eine Task aus dem zweiten Design in das erste hineinkopiert. Hier werden alle Routen wieder hergestellt, diesmal aber werden die vier Ausgangsleitungen des Zähler-Tasks mit den Eingängen des Komparators verbunden (siehe Abb. 7-7).

Auch diese Variante funktionierte. Es zeigte sich aber, dass man die Verbindungen

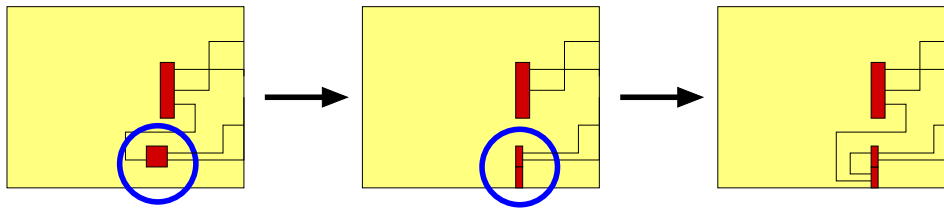


Abbildung 7-8  
Funktion des Programms *Test3.java*

in der "richtigen" Reihenfolge wieder herstellen musste, da sich der Autorouter ansonsten selber den Weg zubaute.

Wir haben danach auch noch ausprobiert, ob es möglich ist, einen der beiden Tasks vor dem Wiederverbinden auch noch an eine andere Stelle zu verschieben. Dazu überprüft das Programm mittels der Funktion `IsInArea()`, ob ein bestimmter Pin innerhalb des verschobenen Gebietes liegt. Falls dies der Fall ist, wird der Pin automatisch an die neue Stelle verschoben, damit beim Verbinden alle Netze wieder korrekt hergestellt werden.

Je weiter weg vom rechten Rand die Tasks aber platziert werden sollten, desto grösser wurde die Wahrscheinlichkeit, dass der JRouter nicht alle Verbindungen herstellen konnte. Scheinbar bereiten ihm längere Routen mehr Mühe, brauchbare Wege zu finden.

Der Aufruf des Programms *test2.java* entspricht demjenigen von *test1.java*.

### 7.5.3 Footprint-Transformationen

Mit einem dritten Programm schliesslich sollte eine CLB-Transformation durchgeführt werden. Als Ausgangs-Design diente uns dabei das Resultat des Programms *test2.java*. Zuerst mussten wiederum alle Routen entfernt werden, die zu den betroffenen CLBs führen.

Der wichtigste Schritt ist nun das Verschieben eines CLB des Komparators, so dass der Footprint eine andere Form erhält. Darauf haben wir die Verbindungen an der neuen Stelle wieder hergestellt (siehe Abb. 7-8). Hier gab es wiederum etliche Probleme mit dem Routing, aber schliesslich gelang die CLB-Task-Transformation des Komparators mit dem Programm *Test3.java*.

Schwieriger war es beim Zähler. Er enthält eine sogenannte Carry-Chain. Das ist eine Kette von schnellen Verbindungen zwischen übereinanderliegenden CLBs, die oft für Counter zur Übertragung des Carry-Bits benutzt wird. Diese kann man nicht auftrennen. Deshalb ist es nicht möglich, einzelne CLBs des Zählers zu verschieben, er kann nur als Ganzes verschoben werden.

Beim Versuch, das Programm auch auf den IP-Stack anzuwenden, gab es sehr viele Schwierigkeiten. Der Autorouter schaffte es nie, alle Verbindungen wieder aufzubauen. Ausserdem zeigte sich, dass der CE-Eingang der CLB-Slices sich nicht mit

dem Autorouter anschliessen liess. Leider blieb uns nicht mehr genügend Zeit, diesen Problemen auf den Grund zu gehen.

## 7.6 Resultate / Fazit

Mit JBits kann ein Bitstream auf viele Arten manipuliert werden. Insbesondere sind damit prinzipiell auch Footprint-Transformationen möglich. In der Realität sind bisher aber erst relativ kleine Änderungen durchführbar, denn es passiert oft, dass nachdem einige Verbindungen geroutet wurden, keine weiteren Verbindungen mehr hergestellt werden können. Es kann dadurch nur ein Teil eines Tasks angeschlossen werden, wodurch er insgesamt nicht funktionsfähig ist. Das Problem ist, dass der Routing-Algorithmus von JRoute zuwenig intelligent ist. Er legt immer nur eine Route nach der anderen. Dadurch kann es passieren, dass er sich später benötigte Wege für andere Verbindungen verbaut. Der richtige Ansatz wäre, dass JRoute alle nötigen Verbindungen gleichzeitig herzustellen versucht und dabei eine möglichst optimale Lösung anstrebt, so dass alle Routen verbunden werden können. Hier wäre Xilinx gefordert, da für die Implementierung der benötigten Algorithmen eine sehr genaue Kenntnis der internen Strukturen des FPGAs nötig ist und recht ähnliche Software bereits für das Routing in den Standard-Tools vorhanden ist.

Eine grosse Hilfe wäre es auch, die für das Routing benutzte Fläche einschränken zu können. So könnte man besser vorgeben, welchen Footprint ein bestimmter Task beanspruchen darf. Dies wäre auch nützlich, um gewisse Flächen im FPGA von jeglichen Routen frei zu halten. Leider kann man bei den Foundation-Tools nur für die CLBs Area-Constrains vorgeben. In JBits gäbe es vielleicht eine Möglichkeit, mit Hilfe des ResourceFactory-Objekts gewisse Bereiche des FPGAs als benutzt zu markieren, so dass der Autorouter sie nicht benützt.

Wir haben auch einen Fehler in JBits entdeckt. Bei gewissen Routen, die am Rande des FPGAs verlaufen, findet JBits bei einem trace nur einen Teil der Verbindung. Man kann ebenfalls nur diesen Teil mit unroute entfernen. Dadurch liessen sich mehrmals Verbindungen nicht entfernen. Ein Beispiel, dass diesen Fehler aufzeigt, ist auf der CD im Verzeichnis `Diplomarbeit/Reconfig/JBits/bugreport` abgelegt.

Bei den Task-Transformationen muss ausserdem auf ein grundsätzliches Problem geachtet werden: Bei jeder Transformation wird zum einen Logik verschoben, zum anderen muss aber auch das Routing angepasst werden. Dies kann die Laufzeit der betroffenen Signale verlängern, was wiederum zur Folge hat, dass die maximale Betriebsfrequenz verringert wird. Deshalb muss man, wenn man Footprint-Transformationen anwenden will, ganz besonders darauf achten, dass das Timing des Designs nicht kritisch ist. Es müssen genügend Timing-Reserven vorhanden sein. Ansonsten kann die zulässige Maximalfrequenz überschritten werden und dadurch wird das Design nur mehr fehlerhaft laufen. Bei unseren kleinen, unkritischen Tasks sind zwar in dieser Hinsicht keine Schwierigkeiten aufgetreten, man muss diesen Aspekt jedoch bei grösseren Designs im Auge behalten.

Zusammenfassend gesehen ist JBits ein brauchbares Paket zum Bearbeiten von Bitstreams, es ist aber nötig, dass die Entwickler von Xilinx weitere Verbesserungen

vornehmen. Beim heutigen Stand ist es noch nicht möglich, Transformationen auf grössere Designs anzuwenden, wo wirklich nutzbare Anwendungen möglich wären.

# 8

## *Ausblick und Dank*

### *8.1 Ausblick*

#### *8.1.1 Software*

Auf Seite der Software sind einige Erweiterungen denkbar:

Eine Möglichkeit ist die Integration des Service-Discovery-Protokolls.

Aufbauend auf unserer Demonstrations-Applikation können ausserdem die in Kapitel 1.1 erwähnten Dienste implementiert werden.

#### *8.1.2 E2B-Bridge*

##### *8.1.2.1 Konfiguration mit dem Dynamic Host Configuration Protocol*

Die Benutzerfreundlichkeit der E2B-Bridge könnte durch DHCP stark erhöht werden. So müsste der Benutzer keine manuelle Netzwerk-Konfiguration im VHDL-File mehr vornehmen, sondern die Bridge würde sich automatisch von einem DHCP-Server im Netz eine gültige IP-Adresse holen.

##### *8.1.2.2 Audio-Übertragungen*

In der Aufgabenstellung wird auch die Integration eines Audio-Modules in die E2BB vorgeschlagen. Wir haben dazu einige Tests durchgeführt. Laut Angaben in den Datenblättern der ROK-Module (siehe Kapitel 2.2) sollten die für Audio-Übertragungen nötigen SCO-Verbindungen möglich sein. Wir haben aber festgestellt, dass der Bluez-Stack damit noch nicht richtig umzugehen weiss.

Daraufhin haben wir versucht, über das RAW-Interface direkt HCI-Kommandos an die Module zu senden, um eine Verbindung aufzubauen. Das Resultat war, dass zwar eines der Module eine Verbindung aufbaute, das andere davon jedoch nichts mitbekommen hatte. Nur das erste zeigte eine aktive SCO-Verbindung an.

Ausserdem haben wir festgestellt, dass beide Module nach diesen Kommandos den PCM-Clock eingeschaltet hatten. An den Datenleitungen PCMin und PCMout regte

sich jedoch nichts und auch durch Anlegen von Signalen an PCMin änderte sich nichts daran.

Deshalb haben wir die Versuche mit Audio-Verbindungen aufgegeben. Es müssen wohl noch einige Versions-Sprünge bei Stack und Modulen abgewartet werden, bis Hard- und Software zusammen funktionieren.

### *8.1.2.3 Benutzung des FPGA-Boards als Endgerät für Audioübertragungen*

Das FPGA-Board könnte als eigenständiges Endgerät für Bluetooth-Audioübertragungen verwendet werden. Dies ist eine Anwendung, bei der die Implementierung einer reduzierten L2CAP-Schicht in Hardware mit verträglichem Aufwand machbar und sinnvoll wäre.

## *8.1.3 Footprint-Transformationen*

Auf diesem Gebiet muss noch viel Grundlagenarbeit geleistet werden. Deshalb gibt es viele Aspekte, die noch näher untersucht werden könnten. Es gibt ausserdem einen Bedarf nach besseren Tools für die Task-Manipulation und die Visualisierung und Simulation von rekonfigurierbaren Umgebungen.

Besonders interessant zu untersuchen wäre das präemptive Task-Scheduling. Dabei geht es nicht nur darum, einen Task an eine freie Stelle zu platzieren und dort laufen zu lassen. Vielmehr kann hier bei Bedarf ein bestimmter Task mit höherer Priorität das Auslagern eines solchen mit niedrigerer Priorität bewirken, noch bevor dieser fertig ausgeführt wurde. Dabei muss der Zustand des zu entfernenden Tasks natürlich zuerst gesichert werden, so dass er bei der Wiederherstellung auch wieder genau rekonstruiert werden kann.

Die Zukunft wird zeigen, was alles machbar ist. Wir sind aber überzeugt, dass trotz all der Probleme, die es noch gibt, schon bald viele neue Anwendungen von rekonfigurierbarer Logik im praktischen Einsatz stehen werden, da sie viele Vorteile bieten und auf diesem Gebiet intensiv geforscht wird.

## *8.2 Dank*

Last, but not least möchten wir an dieser Stelle folgenden Personen danken, die uns tatkräftig unterstützt haben:

- Unseren Betreuern Jan Beutel und Herbert Walder
- Marco Wirz und Matthias Dyer für die nützlichen Gespräche über VHDL und JBits.
- Pierre-André Jacquod für seine Einstiegshilfe in Java.
- Peter Fercher und Egon Burgener für die Zusammenarbeit zur Überwindung der Bluetooth-Stolpersteine.
- Boris Lutz und Claudio Jecker für ihre nützlichen Tipps zu CVS.
- Den immer hilfsbereiten TIK-Supportern Daniel Ott, Thomas Steingruber, Felix Etter und Milan Tadjan.

- Christian Plessl für seine Hilfe mit den VHDL-Tools.
- Barbara und Fleur für die tatkräftige Unterstützung.
- Unseren Mitstudenten für die gute Stimmung im G69.
- Allen anderen, die uns geholfen haben.

Herzlichen Dank!



# A

## Howto

In diesem Kapitel werden alle Aspekte erläutert, die für eine Inbetriebnahme oder Weiterführung unserer Arbeit von Nutzen sein können.

### A.1 Struktur der beiliegenden CD

Die beigelegte CD hat folgende Struktur:

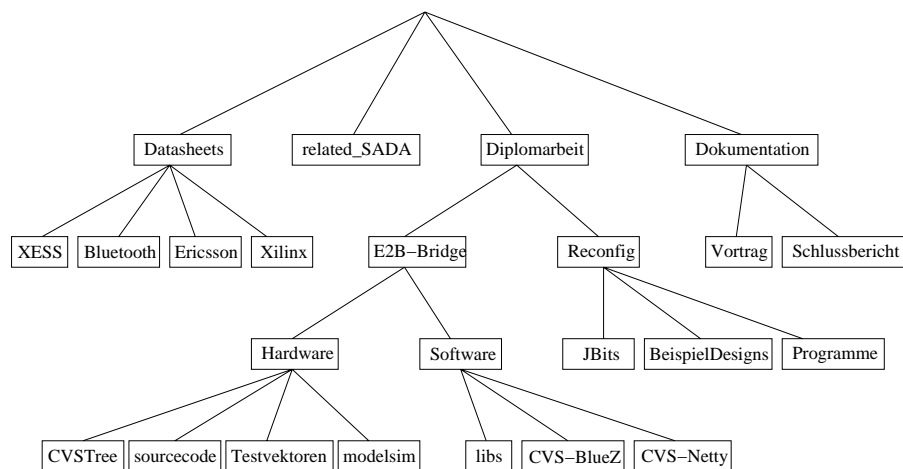


Abbildung A-1  
File-Struktur der beigelegten CD

### A.2 Beschreibung der ausführbaren Programme

Dieses Kapitel beschreibt die ausführbaren Programme auf der beigelegten CD. Dabei wird unterschieden zwischen den Bluez-Programmen, die zum Teil erweitert wurden und den selbargeschriebenen Programmen.

## A.2.1 Bluez - Programme

Die Programme basieren auf der offiziellen Bluez-CVS-Version vom 9. November 2001 (siehe [4]). Dabei ist zu beachten, dass einige Programme an unser Einsatzgebiet angepasst wurden und sich somit der Gebrauch von den offiziellen Versionen unterscheiden kann. Eine sehr gute Dokumentation der aktuellen Bluez-Version ist auf [5] erhältlich.

Die folgenden Programme sind auch auf der beigelegten CD enthalten:

- `bluez/daemons/hcid`

Initialisiert alle UART-Devices, die im config.file definiert, und initialisiert alle mit hciattach gestarteten Devices.

```
Usage: hcid [-n not_daemon] [-f config file]
Bsp  : hcid
```

- `bluez/kernel/update_kernel`

Da Bluez schon standardmässig vom Linux-Kernel unterstützt wird, müssen die Include-Files auf den neusten Stand gebracht werden, falls eine aktuellere Bluez-Version benützt werden soll. Dies geschieht mittels dieses Skripts.

- `bluez/tools/bluepin`

Für Passwordeingaben

- `bluez/tools/hciattach`

Startet UART-Devices.

```
Usage: hciattach <tty> <type | id> [speed] [flow]
Bsp  : hciattach ttyS0 ericsson 57600 flow
```

- `bluez/tools/hciconfig`

Tool zum Konfigurieren des Bluetooth-Moduls.

```
Usage: hciconfig hciX [command]
```

wobei [command] eines der folgenden Kommandos sein kann:

up	Open and initialize HCI device
down	Close HCI device
reset	Reset HCI device
rstat	Reset statistic counters
auth	Enable Authentication
noauth	Disable Authentication
encrypt	Enable Encryption
noencrypt	Disable Encryption
piscan	Enable Page and Inquiry scan
noscan	Disable scan
iscan	Enable Inquiry scan
pscan	Enable Page scan

inq	[length]	Inquire other devices
ptype	[type]	Get/Set default paket type
lm	[mode]	Get/Set default link mode
lp	[policy]	Get/Set default link policy
conn		Display current connections
features		Display device features
name	[name]	Get/Set local name
class	[class]	Get/Set class of device
version		Display version information
writeBD	[addr]	Write Address of ROK 107 (beta status)
rsssi		read RSSI of conn with handle 0x0001

Bsp: hciconfig hci0 inq 10

- **bluez/tools/l2ping**

**Bluetooth Ping, schickt ACL-Ping-Requests**

Usage: l2ping [-S source addr] [-s size]  
                  [-c count] [-f] <bd\_addr>

Bsp : l2ping 01:02:03:04:05:06

- **bluez/tools/l2test**

**Verbindungstest, sendet die Paket-Länge und eine Sequenz-Nummer und füllt die Pakete mit 0x7f. Diese Angaben werden auf der Empfangsseite ausgelesen und falls sie sich vom Erwarteten unterscheiden, wird eine Fehlermeldung ausgegeben.**

**Erweiterung: Beim Sender kann zusätzlich noch eine *Verzögerungszeit t* nach jedem geschickten Paket aktiviert werden.**

Usage: l2test <mode> [-b bytes] [-S bd\_addr] [-P psm]  
                  [-I imtu] [-O omtu] [-M] [bd\_addr]

Modes: -d dump (server)  
          -c reconnect (client)  
          -m multiple connects (client)  
          -r receive (server)  
          -s send (client) -t usec

Bsp: l2test (server)  
      l2test -s 01:02:03:04:05:06

- **bluez/tools/scotest**

**Zum Testen von SCO-Verbindungen, gleiche Vorgehensweise wie l2test, öffnet aber einen SCO-Socket anstatt eines ACL-Sockets.**

Usage: scotest <mode> [-b bytes] [bd\_addr]

Modes: -d dump (server)  
          -c reconnect (client)  
          -m multiple connects (client)

```
-r receive (server)
-s send (client)
```

- **hcidump/hcidump**

Stellt tcpdump-ähnlich alle HCI-Pakete dar, die vom Bluez-Stack verarbeitet werden. Die Pakete werden soweit möglich in menschenlesbare Form umgewandelt. Falsche Pakete werden ignoriert. (Für falsche Pakete siehe DUMP-Ausgaben in /var/log/messages)

```
Usage: hcidump [OPTION...] [filter]
-i, --device=hci_dev      HCI device
-s, --snap-len=len       Snap len (in bytes)
-r, --read-dump=file      Read dump from a file
-w, --save-dump=file      Save dump to a file
-a, --ascii               Dump data in ascii
-h, --hex                 Dump data in hex
-R, --raw                 Raw mode
-?, --help                Give this help list
    --usage                Give a short usage message
-V, --version              Print program version
```

```
Bsp: hcidump -i hci0 -h
```

- **hcidump/hcitool**

Sendet RAW-Pakete zu einem BT-Modul.

```
Usage: hcitool [-i hciX] OGF OCF param...
```

where

```
OGF is OpCode Group Field (00-3F),
OCF is OpCode Command Field (0000-03FF),
param... are parameters.
```

Each parameter is a sequence of bytes.

Bytes are entered in hexadecimal form without spaces, most significant byte first.

The size of each parameter is determined based on the number of bytes entered.

```
Bsp : hcitool -i hci0 01 0001 338B9E05C8 (inq)
```

- **rfcomm/rfcomm**

RFCOMM Dämon, baut eine ppp-Verbindung auf.

Usage:

```
Server: rfcomm <-s> [-n] [-f file] [-P psm]
Client: rfcomm [-n] [-f file] [-P psm] [-t timeout]
        <session> <server bd adress>
```

Bsp :

```
Server: rfcomm -n -f server.conf
Client: rfcomm -n -f client.conf tiger 01:02:03:04:05:06
```

**Files:**

- **server.conf:**

```
options {
    psm 3;          # Listen on this psm.

    ppp            /usr/sbin/pppd;
    ifconfig       /sbin/ifconfig;
    route          /sbin/route;
    firewall       /sbin/ipchains;
}

tiger {
    channel 1;
    up {
        ppp "noauth 10.1.0.1:10.1.0.2";
    };
}
```

- **client.conf:**

```
options {
    psm 3;          # Listen on this psm.

    ppp            /usr/sbin/pppd;
    ifconfig       /sbin/ifconfig;
    route          /sbin/route;
    firewall       /sbin/ipchains;
}

tiger {
    channel 1;
    up {
        ppp "57600 noauth";
    };
}
```

## A.2.2 Programmpaket Netty

Dieses Kapitel enthält eine Erklärung aller selber geschriebenen Programme, die wir während unserer Arbeit programmiert haben:

- `tools/bin2hex.pl`

Wandelt eine binäre Datei in eine Hex-Datei um. Das Hex-Format kann direkt vom Testbench der E2BB und unseren Test-Tools verarbeitet werden. Damit lassen sich identische Test-Pakete sowohl im Testbench wie auch auf dem realen Netzwerk generieren.

- `tools/hex2udp.pl`

Verschickt in einer Hex-Datei enthaltene Pakete mittels UDP-Protokoll an die angegebene IP-Adresse. Dient zum Testen des IP-Stacks.

- `tools/hex2eth.pl`

Verschickt in einer Hex-Datei enthaltene Pakete als Raw-Ethernet-Paket. Dient zum Testen des IP-Stacks.

- `tools/tx`

Tool zum Generieren von Netzwerkpaketen.

```
usage: ./tx [-i interface] -p RAW|ETH|TCP|UDP
        -s source -d destination
```

```
Bsp : echo hallo| ./tx -i eth1 -p ETH -d ff:ff:ff:ff:ff:ff
```

```
Bsp : echo hallo| ./tx -i eth1 -p TCP -d 10.0.0.1.1000
```

- `netty`

Softwarepaket für die Kommunikation mit einer Ethernet-Bluetooth-Bridge.

```
Usage:  ./netty -s (server - listen)
        ./netty -c (client - connect [default])
        ./netty -u /dev/pty (user interface through pty)
        ./netty -d drop_rate [percent]
        ./netty -t timeout [usec]
```

```
Bsp :  ./netty -d 10 -t 100000
```

Nach dem Starten wartet das Programm auf weitere Benutzereingaben, wie zum Beispiel den Befehl, eine Verbindung aufzubauen.

```
usage:  (add connection)      -a IP->IP:PORT
        (quit)                -q
        (debug)               -d closed2syn_sent
                               -d syn_sent2established
                               -d close (established2fin_wait1)
```

```
Bsp :  -a 10.0.0.3->10.0.0.4:2000
```

- `qt_simple/netty_simple.qt`

Grafische Oberfläche zu Netty. Der Verbindungsaufbau der ersten Verbindung kann mittels eines Formulars konfiguriert werden.

```
./netty_simple.qt
```

- `tools/demo_get_rssi/rssi_server`

Server für RSSI-Messungen, wartet auf ACL-Verbindungen auf PSM 11 und liefert, sobald eine Verbindung zustande kommt, die RSSI-Werte über die ACL-Verbindung zurück.

```
Usage: ./rssi_server -a <locale address> -d <hciX>
Bsp  : ./rssi_server -a 01:02:03:04:05:06 -d hci0
```

- `tools/demo_get_rssi/demo_get_rssi.qt`

Clientprogramm für RSSI-Messungen. Sucht mittels Inquiries nach vorhandenen Bluetooth-Geräten. Danach versucht es mit allen gefundenen Geräten, die nicht in der Ignore-Liste sind, auf PSM 11 eine Verbindung aufzubauen. Die empfangen RSSI-Werte werden grafisch dargestellt.

```
Usage: ./demo_get_rssi.qt
```

## A.3 Inbetriebnahme

### A.3.1 Bluez

Um den Bluez-Stack in Betrieb zu nehmen, folge man dem Bluez-Howto (das in der entsprechenden Version auf der beigelegten CD zu finden ist). Falls ein aktueller Bluez-Stack verwendet werden soll, der aber möglicherweise nicht kompatibel zu unseren Anpassungen ist, ist eine Online-Variante des Bluez-Howto [5] vorhanden.

Ein häufiger Fehler ist, dass vergessen wird, das Skript `update_kernel` auszuführen.

Ist Bluez installiert, müssen die Kernelmodule gestartet werden. Man benötigt mindestens ein Modul für die Treiberunterstützung des HCI-Transport-Layers (z.B. UART → `hci_uart` oder USB → `hci_usb`), ein weiteres Modul für den Bluez-Core (in unserer Version hat es den Namen `bluez`). Des Weiteren müssen die Module für die Protokolle geladen werden (`l2cap` und ev. `sco`). Ebenso muss `hcid` gestartet werden, falls die angeschlossenen Geräte automatisch initialisiert werden sollen.

### A.3.2 Netty

Ist der lokale Betrieb von Bluetooth-Modulen möglich (`hciconfig` gibt die Bluetooth-Adresse der angeschlossenen Bluetooth-Hardware zurück), kann damit begonnen werden, Netty in Betrieb zu nehmen. Netty muss als `root` ausgeführt werden, da es sonst die verwendeten Schnittstellen nicht öffnen darf. Die möglichen Kommandozeilen-Optionen sind in Kapitel A.2.2 ausführlich beschrieben.

### A.3.3 E2B-Bridge

Für die Inbetriebnahme der E2B-Bridge müssen auf dem Host zuerst die XSTools von XESS installiert werden. Sie sind auf [11] für Windows und Linux erhältlich.

Den Systemtakt des Boards muss man auf 16 MHz einstellen. Dazu braucht man einen externen Oszillator, der an den mittleren Pin von Jumper 36 auf dem Board angeschlossen werden muss.

Anschließend muss das CPLD neu programmiert werden. Zuerst wechselt man mittels `cd` ins Verzeichnis `Diplomarbeit/E2B-Bridge/Hardware`. Danach lädt man die Datei `cpld.svf` mit Hilfe des Programmes `xsload` auf das Board hinunter. Dazu tippt man in der Shell:

```
cd Diplomarbeit/E2B-Bridge/Hardware
xsload cpld.svf
```

Ein Neustart des Boards aktiviert daraufhin diese Konfiguration.

Nun muss noch der FPGA konfiguriert werden. Dies geschieht wiederum mit dem Programm `xsload`, der Bitstream ist in der Datei `fpgala.bit` gespeichert:

```
xsload fpgala.bit
```

Die E2BB ist danach einsatzbereit. Sie sollte nun auf Ping-Anfragen an ihre Standard IP-Nummer 10.0.0.4 antworten. Wenn dies funktioniert, kann man mit Netty eine Verbindung auf Port 2000 der Bridge aufbauen. Der serielle Port arbeitet standardmässig mit 57600 bit/s.

Für Angaben zum Ändern von IP-Adresse und RS232-Bitrate sei auf die entsprechenden Kapitel 5.3.1 und 5.4.1 verwiesen.

### A.3.4 Rekonfiguration

Zum Nachvollziehen unserer Versuche mit *JBits* benötigt man ein installiertes *Java Runtime Environment* Version 1.2.2. Danach muss *JBits* Version 2.8 installiert werden. Zum Testen auf dem FPGA braucht man ein XESS XSV-Board sowie die XESS-Tools.

Die zugehörigen Programme sind auf der CD im Verzeichnis `Diplomarbeit/Reconfig/Programme` zu finden. Man darf nicht vergessen, bei Bedarf die `CLASSPATH`-Environment-Variable richtig zu setzen.

Die drei Programme `test1`, `test2` und `test3` können danach im Java-Environment gestartet werden:

```
java myself.test.test1 <bitstream1.bit> <bitstream2.bit> <bitstreamout.bit>
```

Die beiden Ausgangs-Designs sind auf der CD im Verzeichnis `Diplomarbeit/Reconfig/BeispielDesigns` unter den Namen `project.bit` und `project2.bit` zu finden. An ihnen lassen sich die in Kapitel 7.5 beschriebenen Veränderungen durchführen. Das Programm `xsload` dient wiederum zum Herunterladen der Bitstreams auf das FPGA-Board.

# B

## Dokumentation der Software Netty

### B.1 File - Struktur

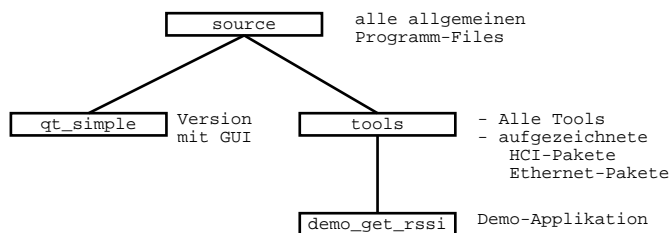


Abbildung B-1  
Verzeichnis-Struktur  
von Netty

In den folgenden Tabellen kann eine kurze Beschreibung der wichtigsten Files gefunden werden:

Verzeichnis <code>./tools/demo_get_rssi</code>	
<code>demo_get_rssi.ui</code> <code>moc_demo_get_rssi_impl.cpp</code> <code>Makefile.qt</code>	Mit qt generierte Files
<code>demo_get_rssi.pro</code> <code>demo_get_rssi_impl.h</code> <code>demo_get_rssi_impl.cpp</code> <code>Makefile</code>	Qt-Files  <code>make header</code> (generiert <code>demo_get_rssi_impl.h</code> ) <code>make impl</code> (generiert <code>demo_get_rssi_impl.cpp</code> ) <code>make moc</code> (generiert moc-Files) <code>make rssi_server</code> (compiliert den Server) <code>make all</code> (compiliert den Client ( <code>demo_get_rssi.qt</code> ))
<code>bluetooth.cpp</code>	Bluetooth Library
<code>main_demo_get_rssi.qt.cpp</code>	Main-File für den Client
<code>main_demo_get_rssi.qt</code>	Client Application
<code>rssi_server.c</code>	Main-File für den Server
<code>rssi_server</code>	Server Application

Verzeichnis <code>./qt_simple</code>	
add_connection.ui moc_add_connection_impl.cpp debugging_options.ui moc_debugging_options_impl.cpp Makefile.qt	Mit qt generierte Files
netty_simple.pro add_connection_impl.h add_connection_impl.cpp debugging_options_impl.h debugging_options_impl.cpp	Qt-Files
Makefile	make header (generiert *_impl.h) make impl (generiert *_impl.cpp) make moc (generiert moc-Files) make all (compiliert den Client)
netty_simple.qt.cpp	Main-File für Applikation mit GUI
netty_simple.qt	Applikation mit GUI
Verzeichnis <code>./</code>	
netty.h	Alle globalen Defines und Includes
hcilib.h	Bluetooth-spezifische Defines
tlist.c	Library für Timer-Liste (Doppelt verlinkte und geordnete Liste)
llist.c	Library für Listen
netlib.c	Library für Netzwerkzugriff
init.c	Initialisierung, Parsen der Optionen
debug.c	Debugging
conn_management.c	Verbindungsauf- und abbau, Kommandozeilen User-Interface
states.c	Überprüfen auf Zustandsübergänge gemäss dem aktuellen Zustand des MC-Protokolls
state_transitions.c	Aktionen bei Verbindungsübergängen des MC-Protokolls
update_states.c	Überwachung des MCP-Zustands
pty.c	Kommunikation mit dem seriellen Treiber von Bluez
hcilib.c	Library zum Parsen von HCI-Paketen
linkfd.c	Callback-Handler, wenn neue Daten vom Netzwerk oder Bluez-Stack ankommen
select.c	Library für select-Aufrufe
Makefile	make all (generiert netty)
netty.c	Main-File
netty	Applikation

## B.2 Konfigurations-Möglichkeiten im File netty.h

In Tabelle B.2 werden die Bedeutungen der verschiedenen #defines im File netty.h erklärt.

## B.3 Compilieren von Netty

Damit die Programme gemäss der Beschreibung der Makefiles (siehe Kapitel B.1) compiliert werden können, müssen folgende Punkte sichergestellt werden:

- Bluez: Entweder muss die Bluez-Version der CD verwendet werden, oder aber die Header-Files in bluez/include müssen angepasst werden. Ebenso muss der Wrapper an das aktuelle hciattach angepasst werden (siehe vor allem netty.h)
- Qt: Falls die Demoapplikation oder Netty mit dem GUI verwendet werden soll, muss Qt installiert sein.

<b>#defines</b> (Standardwert)	Erklärung
<b>DEBUG</b> (0)	Debugging Level, wobei 9 das höchste Level ist bei dem alle Messages ausgegeben werden.
<b>DUMP</b> (9)	Level der Datenausgaben
<b>MAX_COMMAND</b> (100)	Maximale Anzahl der Buchstaben eines Benutzer-Interface-Kommandos
<b>MAX_FRAME_SIZE</b> (1024)	Maximale Anzahl Bytes eines MCP-Frames
<b>SERVER.TTY</b> ("/dev/ttyS0")	Falls <i>Netty</i> eine Bluetooth-Bridge emuliert, kann mit diesem Parameter die serielle Schnittstelle angegeben werden, an welche das Bluetooth-Modul angeschlossen ist.
<b>SERVER.SPEED</b> (B57600)	Baud-Rate des Bluetooth-Moduls
<b>WIN_SIZE</b> (16)	Anzahl der Empfangs- und Sendepuffer
<b>TIMEOUT_SEC</b> (0)	Timeout: Anzahl ganzer Sekunden
<b>TIMEOUT_USEC</b> (100000)	Timeout: Anzahl $\mu\text{sec}$
<b>HCIATTACH</b> ("hciattach")	Kommandozeilen-Befehl um ein neues Bluetooth-Gerät an den BlueZ-Stack anzuhängen.
<b>LOG_ERR</b> (2) = StdErr	File-Deskriptor, in den die Errors geschrieben werden.
<b>LOG_DBG</b> ("netty.dbg")	Datei, in welche die Debugging-Messages geschrieben werden sollen.
<b>FD_USER</b> (0) = StdIn	File-Deskriptor, mit dem die User-Interaktion stattfinden soll.

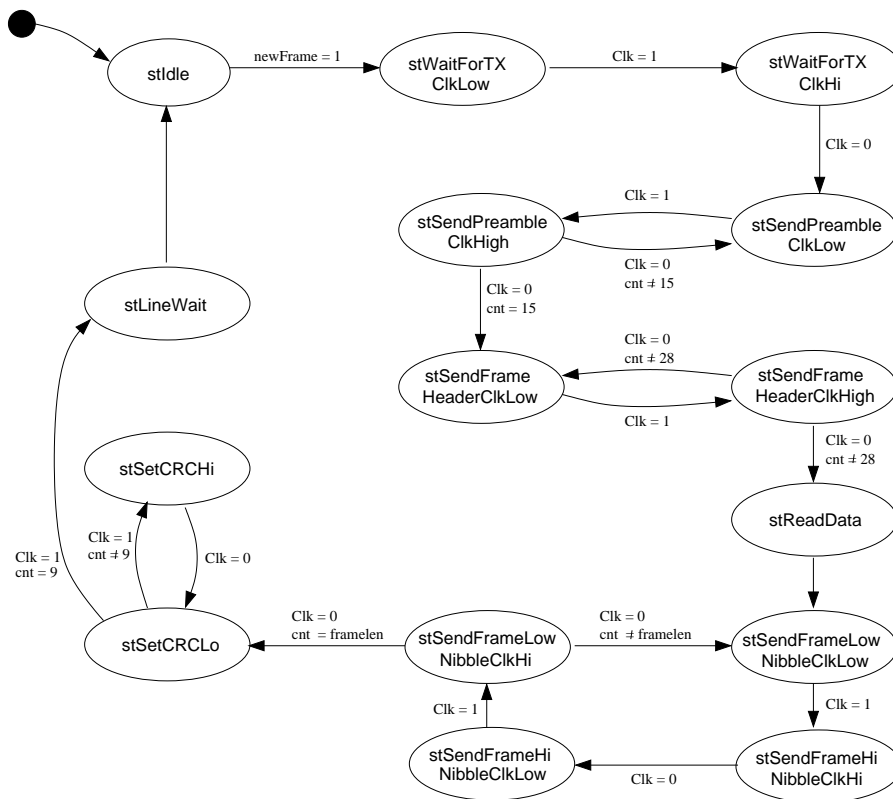
Tabelle B-1: Erklärung der Konfigurations-Möglichkeiten mittels #defines im File netty.h



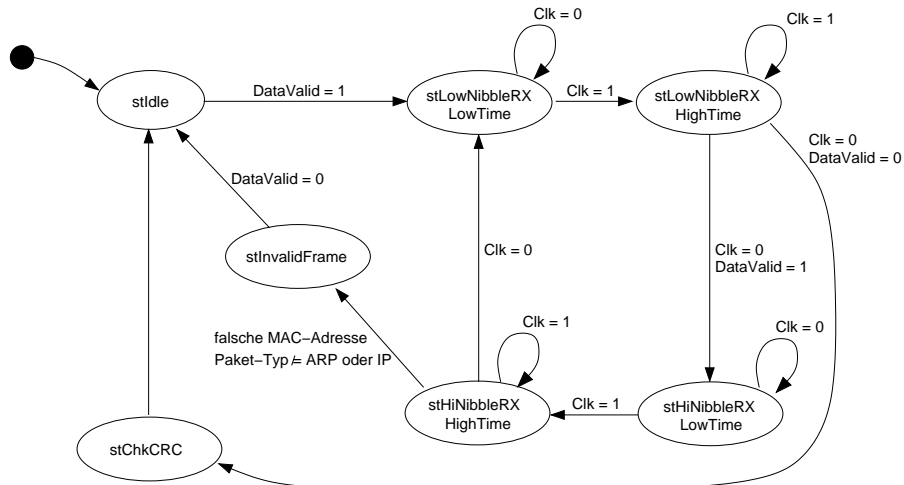
# C

## Zustands-Diagramme des IP-Stacks

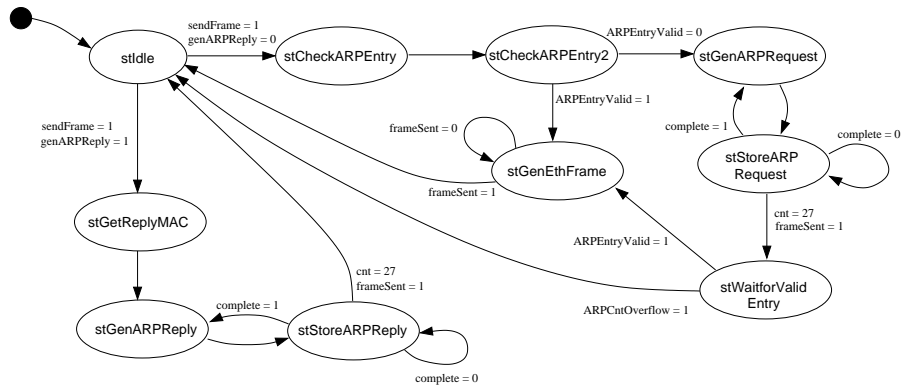
### C.1 Ethernet-Sender (ethernetsnd.vhd)



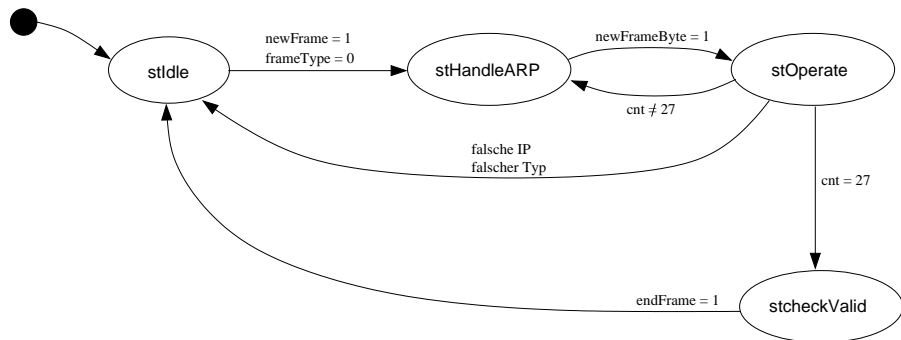
### C.2 Ethernet-Empfänger (ethernet.vhd)



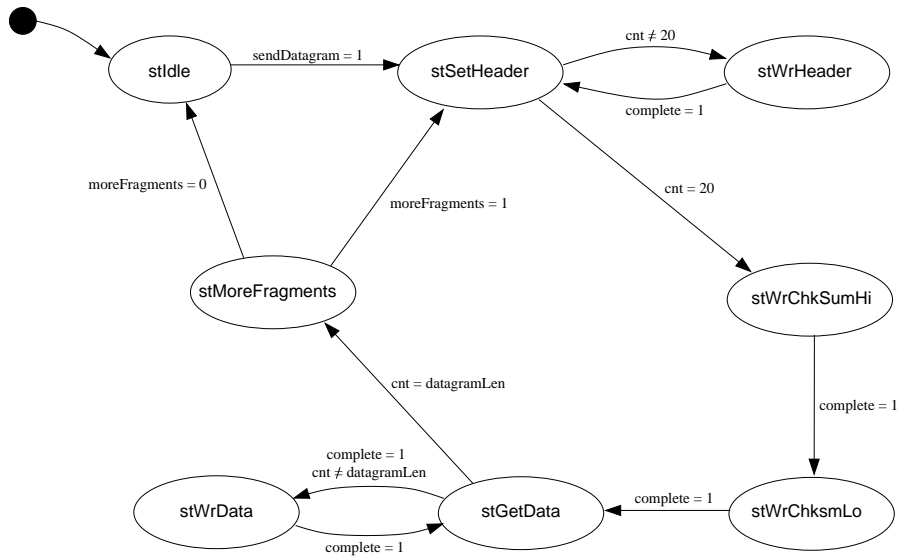
### C.3 ARP-Sender (arpsnd.vhd)



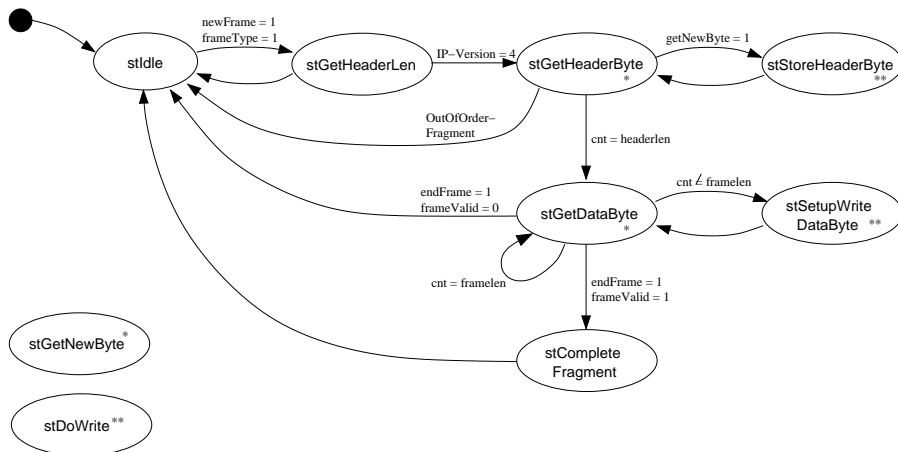
### C.4 ARP-Empfänger (arp3.vhd)



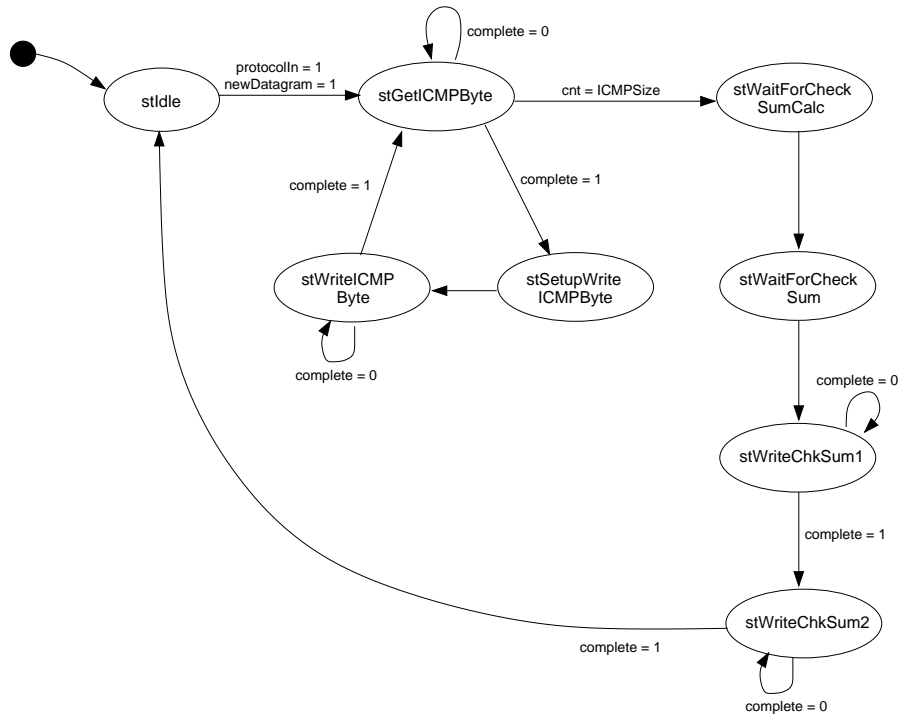
### C.5 Internet-Sender (internetsnd.vhd)



### C.6 Internet-Empfänger (internet.vhd)



### C.7 ICMP (icmp.vhd)



# D

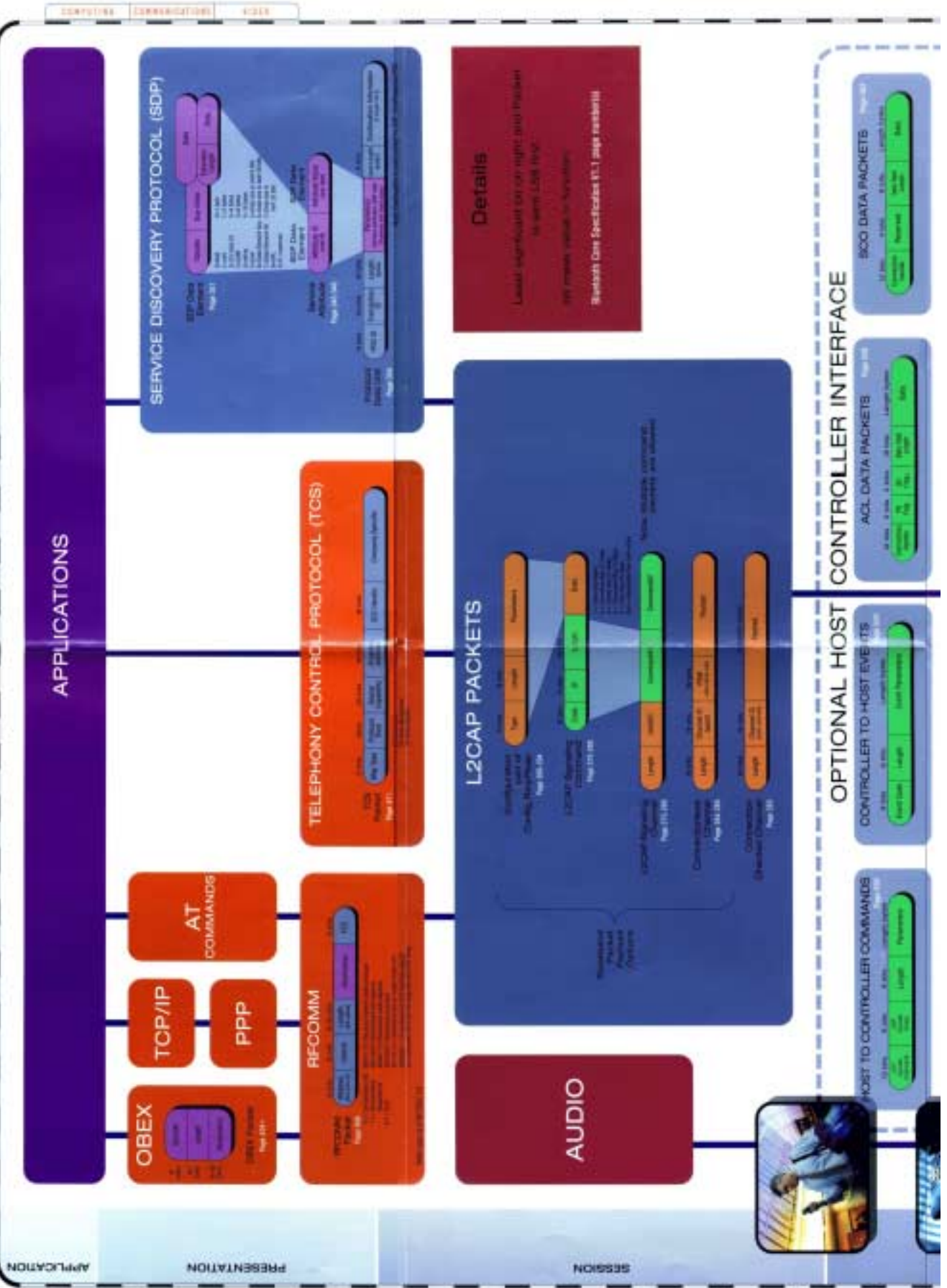
## *Bluetooth*

### *D.1 Grafische Darstellung des gesamten Bluetooth-Stacks*

In der elektronischen Fassung dieses Berichts ist das Bild als zwei hochauflösende A4-Seite eingefügt. Beim Drucken des Berichts müssen diese Seiten auf mindestens A3 ausgedruckt werden, damit alle Beschriftungen auch lesbar sind.

# BLUETOOTH™ PROTOCOL STACK

OSI LAYERS





## D.2 Traces von HCI - Sequenzen

Die folgenden Seiten zeigen die HCI-Sequenzen, die in verschiedenen Szenarien über die UART-Schnittstelle geschickt werden. Die Seitenzahlen beziehen sich auf die Bluetooth-Spezifikation in der Version 1.1 [1], die sich auch auf der beigelegten CD befindet.

Die ersten Traces wurden selber aufgezeichnet, während die Nachfolgenden in Zusammenarbeit mit der Semesterarbeit [9] entstanden und zur Vollständigkeit übernommen wurden (vielen Dank der Gruppe [9]).

### D.2.1 Init Sequence

Init-Sequence (hciattach when hcid is running)

-----

# local uart speed must be 57600!

```
< HCI Command: Ericsson specific speed set          -      see ROK 007 module spec
01  09 FC  01  03
cmd  ogf/ocf plen  set speed
      opcode      to 57600 baud
```

```
> HCI Event: Command Complete(0x0e) plen 4
04  0E  04  08          09 FC  00          -      see ROK 007 module spec
evt  command      plen=4  number of      cmd opcode  status 00=
      complete      hci packets  which caused  success
                                that one can still  this event
                                send to rok
```

# the next few commands are the same as "hciconfig hci0 up"

```
< HCI Command: Read Local Supported Features(0x04|0x0003) plen 0          -      Page 702
01  03 10  00
cmd  opcode      plen
```

```
> HCI Event: Command Complete(0x0e) plen 12          -      Page 744 / 702 / 243
04  0E  0C  08          03 10  00          FF F9 01 00 00 00 00
evt  code  plen  num hci cmd pkts  cmd opcode  status 00 = success  features Page 243 (only first three bytes relevant up till now)
                                see table on page 243
```

```
< HCI Command: Read Buffer Size(0x04|0x0005) plen 0          -      Page 703
01  05 10  00
cmd  opcode      plen
```

```
> HCI Event: Command Complete(0x0e) plen 11          -      Page 704
04  0E  0B  08          05 10  00  A0 02  00  08 00  00 00
evt  code  plen  num hci cmd pkts  cmd opcode  status  acl max len  max sco len  num acl  num sco
                                0x02a0 = 672  sco not supp.  8 pkts max  sco not supp.
```

```
< HCI Command: Read BD ADDR(0x04|0x0009) plen 0          -      Page 706
01  09 10  00
cmd  opcode      plen
```

```
> HCI Event: Command Complete(0x0e) plen 10          -      Page 706
04  0E  0A  08          09 10  00  55 00 00 00 00 00
evt  code  plen  num hci cmd pkts  cmd opcode  status 00:00:00:00:00:55 is address of device
```

```

< HCI Command: Set Event Filter(0x03|0x0005) plen 1 - Page 623
01 05 0C 01 00
cmd opcode plen parameter

> HCI Event: Command Complete(0x0e) plen 4 - Page 623
04 0E 04 08 05 0C 00
evt code plen num hci cmd pkts cmd opcode status = success

< HCI Command: Write Page Timeout(0x03|0x0018) plen 2 - Page 645
01 18 0C 02 00 80
cmd opcode plen 0x8000 page timeout measured in number of baseband slots, 0x8000 * 0.625 msec = 20.48 sec.

> HCI Event: Command Complete(0x0e) plen 4 - Page 645
04 0E 04 08 18 0C 00
evt code plen num hci pkts cmd opcode status success

< HCI Command: Write Connection Accept Timeout(0x03|0x0016) plen 2 - Page 643
01 16 0C 02 00 7D
cmd opcode plen 0x7d00 * 0.625 msec = 20 sec.

> HCI Event: Command Complete(0x0e) plen 4 - Page 643
04 0E 04 08 16 0C 00
evt code plen num hci pkts cmd opcode status success

< HCI Command: Write Scan Enable(0x03|0x001a) plen 1 - Page 647
01 1A 0C 01 03
cmd opcode plen Inquiry enabled, page scan enabled

> HCI Event: Command Complete(0x0e) plen 4 - Page 647
04 0E 04 08 1A 0C 00
evt code plen num hci pkts cmd opcode status success

# additional initial commands which are not included in "hciconfig hci0 up"

< HCI Command: Write Authentication Enable(0x03|0x0020) plen 1 - Page 657
01 20 0C 01 00
cmd opcode plen Authentication disabled (default)

> HCI Event: Command Complete(0x0e) plen 4 - Page 657
04 0E 04 08 20 0C 00
evt code plen num hci pkts cmd opcode status success

# authentication once again? probably a bug in BlueZ

    < HCI Command: Write Authentication Enable(0x03|0x0020) plen 1
    01 20 0C 01 00

    > HCI Event: Command Complete(0x0e) plen 4
    04 0E 04 08 20 0C 00

< HCI Command: Write Encryption Mode(0x03|0x0022) plen 1 - Page 659
01 22 0C 01 00
cmd opcode plen Encryption disabled (default)

> HCI Event: Command Complete(0x0e) plen 4 - Page 660
04 0E 04 08 22 0C 00
evt code plen num hci pkts cmd opcode status success

```

```

< HCI Command: Write Class of Device(0x03|0x0024) plen 3 - Page 662
01 24 0C 03 00 01 00
cmd opcode plen 0x000100 class of device
= Computer (desktop,notebook, PDA, organizers, ... )
(siehe http://www.bluetooth.org/assigned-numbers/baseband.htm)

> HCI Event: Command Complete(0x0e) plen 4 - Page 662
04 0E 04 08 24 0C 00
evt code plen num hci pkts cmd opcode status success

< HCI Command: Change Local Name(0x03|0x0013) plen 248 - Page 640
01 13 0C F8 42 6C 75 65 5A 00 xx ... xx
cmd opcode plen user friendly name of the device, the array has to be 248 bytes always,
if the string is shorter, it has to be null terminated.
-> 0x426C75655A00... = BlueZ

```

Example of a command:

```

01 13 0C F8 42 6C 75 65 5A 00 00 00 10 17 00 40 A4 0A 00 40
D4 03 00 40 C0 69 01 40 05 00 00 00 B8 76 01 40 05 00 00 00
24 88 04 08 01 00 00 00 00 00 68 6D 01 40 AC 5A 70 00
8B 37 01 40 60 F8 FF BF 10 6C 01 40 D8 8A 04 08 8B 37 01 40
70 F8 FF BF 9C E9 04 40 98 74 01 40 01 00 00 00 00 00 00 00
EC 41 05 40 98 74 01 40 8B 37 01 40 90 F8 FF BF 00 00 00 00
00 00 00 00 F8 DA 02 40 FE A2 0C 40 A4 9C 02 40 A0 72 01 40
05 00 00 00 B8 76 01 40 02 00 00 00 D4 A9 02 40 01 00 00 00
00 00 00 00 F4 73 01 40 F8 65 01 40 A0 72 01 40 FE A2 0C 40
50 F8 FF BF 16 D8 00 40 F4 73 01 40 04 00 00 00 38 01 00 00
F8 65 01 40 10 6C 01 40 E8 76 01 40 70 F8 FF BF 16 D8 00 40
64 6D 01 40 08 77 01 40 07 00 00 00 00 00 00 00 90 1F 05 08
07 00 00 00 28 01 00 00 29 00 00 00

```

```

> HCI Event: Command Complete(0x0e) plen 4 - Page 640
04 0E 04 08 13 0C 00
evt code plen num hci pkts cmd opcode status success

```

## D.2.2 RSSI

RSSI-Example

-----

# This command-sequence works with ROK 007 (not multipoint!) and ROK 008

# A connection has to be created first (see l2test for details)

```

< HCI Command: Create Connection(0x01|0x0005) plen 13 - Page 568
01 05 04 0D 04 00 00 00 00 18 00 00 00 00 01
cmd ogf/ocf plen=13 bdaddr= pkt type= page scan page scan clock allow_role_switch
opcode 00:00:00:00:00:04 0x0018 (acl) rep mode R0 scan mode= mand offset master may switch to slave

```

< > ... (see l2test-traces for details)

# then the RSSI of this connection can be measured

```

< HCI Command: Read RSSI (0x05|0x0005) plen 2 - Page 714
01 05 14 02 01 00
cmd ogf/ocf plen=2 0x0001 = handle of the connection which should be
opcode which should be measured

```

```

> HCI Event: Command Complete(0x0e) plen 4
04 0E 04 08 05 14 - Page 714 01 00 03
evt  command plen=4  number of  cmd opcode  status 0x0001 = handle of the  RSSI as signed integer (-128 <= RSSI <= 127)
      complete      hci packets  which caused 00=success  connection which was measured [dB]

```

## D.2.3 Change BT-Address

Change Address

-----

```

# for some (non multipoint) ROK 007/008 (please note that this is not a officially documented HCI-Command, its only working for ROK 007.
# Use it at your own risk only!!)

< HCI Command: Write BT_ADDR (0x3F|0x000D) plen 6 - extracted from the axis-stack
01 0D FC 06 06 05 04 03 02 01 (its a not documented ericsson specific HCI_Command)
cmd opcode plen new address in network order
-> 01:02:03:04:05:06

> HCI Event: Command Complete(0x0e) plen 4
04 0E 01 08 0D FC 00
evt code plen num hci cmd pkts cmd opcode status 00 = success
                                     if you get something different from 00,
                                     then your module supports most likely a different way to change the address

# for some multipoint ROK 007 (please note that this is a trace of the officially distributed Windows-Program (WriteBDADDR.exe).
# Because it does overwrite the flash-memory of the module, the usage is NOT recommended.
# Use it at your own risk only!!!!)

< HCI Command: Read Local Supported Features(0x04|0x0003) plen 0 - Page 702
01 03 10 00
cmd opcode plen

> HCI Event: Command Complete(0x0e) plen 12 - Page 744 / 702 / 243
04 0E 0C 08 03 10 00 FF F9 01 00 00 00 00 00
evt code plen num hci cmd pkts cmd opcode status 00 = success features Page 243 (only first three bytes relevant up till now)

< HCI Command: WriteFlashMemory (0x3F|0x0022)
01 22 FC FF FE 06 11 11 50 11 11 11 00 ... 00 B0 33
cmd opcode new address in network order 224* 0x00
-> 11:11:11:50:11:11

# kind of strange length, but unfortunately we did not found any documentation ;-(
# guessing that FF is the plen as usual, which would not match the actual size...

> HCI Event: Command Complete(0x0e) plen 4
04 0E 04 08 22 FC 00
evt code plen num hci cmd pkts cmd opcode status 00 = success

```

## D.2.4 Inquiry [9]

INQUIRY (hciconfig hci0 inq 5)

-----

```
< HCI Command: Inquiry(0x01|0x0001) plen 5 - Page: 561
01      01 04      05      33 8B 9E 05      C8
cmd      ogf/ocf plen      0x9e8b33 time      max
          opcode          Company ass. before inq num of
                               LAP           is halted responses (=200)

> HCI Event: Command Status(0x0f) plen 4 - Page: 745
04      0F          04      00          07          01 04
evt      command    plen      command is 7 packets  command opcode that is pending
          status          currently pending may be sent to rok

> HCI Event: Inquiry Result(0x02) plen 15 - Page: 728
04      02      0F          01          04 00 00 00 00 00      01      00      00      00 01 00 AF 4B
evt      inq      plen=15      Num_Resp=1      bt address      01: 1 Byte * Num_Resp: Page_Scan_Repetition_Mode (=R1)
          result          00:00:00:00:00:04 00: 1 Byte * Num_Resp: Page_Scan_Period_Mode (=P0)
                               6bytes * Num_Resp 00: 1 Byte * Num_Resp: Page_Scan_Mode (=Mandatory Page Scan Mode)
                               0x000100: 3 Byte * Num_Resp: Class Of Device
                               0x4BAF : 2 Byte * Num_Resp: Clock Offset

> HCI Event: Inquiry Complete(0x01) plen 1 - Page: 727
04      01          01          00
evt      inquiry complete plen=1      success

> HCI Event: Command Status(0x0f) plen 4 - Page: 745
04      0F          04      00          08          00 00
evt      command status plen=4      command      number of      Opcode of command
          currently      pending      hci packets      which caused this event and is pending completion
          that one can still
          send to rok
```

## D.2.5 hci\_config [9]

HCICONFIG

-----

```
[root@pc-3294 /root]# hciconfig hci0 up
< HCI Command: Read Local Supported Features(0x04|0x0003) plen 0 - Page 702
01      03 10      00
cmd      opcode          plen
```

```

> HCI Event: Command Complete(0x0e) plen 12
04 0E 0C 08 03 10 00 - Page 744 / 702 / 243
evt code plen num hci cmd pkts cmd opcode status 00 = success features FF F9 01 00 00 00 00 00
see table on page 243

< HCI Command: Read Buffer Size(0x04|0x0005) plen 0
01 05 10 00 - Page 703
cmd opcode plen

> HCI Event: Command Complete(0x0e) plen 11
04 0E 0B 08 05 10 00 A0 02 00 08 00 00 00
evt code plen num hci cmd pkts cmd opcode status acl max len max sco len num acl num sco
0x02a0 = 672 sco not supp. 8 pkts max sco not supp.
- Page 706

< HCI Command: Read BD ADDR(0x04|0x0009) plen 0
01 09 10 00 - Page 706
cmd opcode plen

> HCI Event: Command Complete(0x0e) plen 10
04 0E 0A 08 09 10 00 55 00 00 00 00 00
evt code plen num hci cmd pkts cmd opcode status 00:00:00:00:00:55 is address of device

< HCI Command: Set Event Filter(0x03|0x0005) plen 1
01 05 0C 01 00 - Page 623
cmd opcode plen parameter

> HCI Event: Command Complete(0x0e) plen 4
04 0E 04 08 05 0C 00 - Page 623
evt code plen num hci cmd pkts cmd opcode status = success

< HCI Command: Write Page Timeout(0x03|0x0018) plen 2
01 18 0C 02 00 80 - Page 645
cmd opcode plen 0x8000 page timeout measured in number of baseband slots, 0x8000 * 0.625 msec = 20.48 sec.

> HCI Event: Command Complete(0x0e) plen 4
04 0E 04 08 18 0C 00 - Page 645
evt code plen num hci pkts cmd opcode status success

< HCI Command: Write Connection Accept Timeout(0x03|0x0016) plen 2
01 16 0C 02 00 7D - Page 643
cmd opcode plen 0x7d00 * 0.625 msec = 20 sec.

> HCI Event: Command Complete(0x0e) plen 4
04 0E 04 08 16 0C 00 - Page 643
evt code plen num hci pkts cmd opcode status success

< HCI Command: Write Scan Enable(0x03|0x001a) plen 1
01 1A 0C 01 03 - Page 647
cmd opcode plen Inquiry enabled, page scan enabled

> HCI Event: Command Complete(0x0e) plen 4
04 0E 04 08 1A 0C 00 - Page 647
evt code plen num hci pkts cmd opcode status success

[root@pc-3294 /root]# hciconfig hc0 down
< HCI Command: Reset(0x03|0x0003) plen 0
01 03 0C 00 - Page 622
cmd opcode plen

```

## D.2.6 l2ping [9]

### D.2.6.1 Client

L2PING (l2ping 11:11:11:50:11:11) - (00:00:00:00:00:04 - pings-> 11:11:11:50:11:11)

```

-----
                here we are !

< HCI Command: Create Connection(0x01|0x0005) plen 13                -      Page 568

01      05 04      0D      04 00 00 00 00 00      18 CC      00      00      00 00      01
cmd      ogf/ocf      plen=13      bd addr      Packet_Type      Page      Page      Clock Off      Allow Role Switch=master
00:00:00:00:00:04 0xcc18      Scan Rep Mod=R0      Scan Mode=Mand      but switch to slave
                                                allowed

> HCI Event: Command Status(0x0f) plen 4                -      Page 745

04      0F      04      00      07      05 04
evt      command      plen=4      status=command      number of further hci      opcode that created evt and is pending
status      pending      cmds that are allowed
                                                to send to rok=7

> HCI Event: Connect Complete(0x03) plen 11                -      Page 730

04      03      0B      00      01 00      04 00 00 00 00 00      01      00
evt      Connection      plen=11      status=      Connectiona      address=      Link_Type=      Encryption_Mode=
Complete      successa Handle (12 Bits) 00:00:00:00:00:04 ACL      disabled

> HCI Event: Command Status(0x0f) plen 4                -      Page 745

04      0F      04      00      08      00 00
evt      command      plen=4      status=pending      now we can again pending command ogf/ocf
status      pass 8 cmd to rok that caused evt

< ACL data: handle 0x0001 flags 0x02 dlen 28                -      Page 555 / 293
L2CAP(s): Echo req: dlen 20

02      01 20      1C 00      18 00 01 00 08      C8      14 00 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58
acl      0x2001      dlen      28 acl data bytes      optional data
      handle/flags      0x001c = 28      L2CAP: 18 00 0x0018 Length = 24 bytes
      01 00 0x0001 Cid (signalling)
      08 0x08 L2cap cmd code (Echo request)
      C8 0xc8 Identifier = 200
      14 00 0x0014 Length of cmd

> ACL data: handle 0x0001 flags 0x02 dlen 28                -      Page 555 / 294
L2CAP(s): Echo rsp: dlen 20

02      01 20      1C 00      18 00 01 00 09      C8      14 00 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58
acl      handle/flags      dlen      28 acl data bytes      optional data
      0x001c = 28      L2CAP: 18 00 0x0018 Length = 24
      01 00 0x0001 Cid (signalling)
      09 0x09 L2CAP cmd code (Echo response)
      C8 0xc8 Identifier = 200 (this is the answer to the above request ;- )
      14 00 0x0014 Length of cmd

```

```

< HCI Command: Disconnect(0x01|0x0006) plen 3 - Page 571
01 06 04 03 01 00 13
cmd opcode plen=3 connection handle reason for disconnect (see Page 571)
0x0001 0x13 = other side terminated connection

> HCI Event: Command Status(0x0f) plen 4 - Page 745
04 0F 04 00 07 06 04
evt code plen=4 status opcode of pending cmd 4 cmds can currently be sent to rok
=pending

> HCI Event: Disconn Complete(0x05) plen 4 - Page 733
04 05 04 00 01 00 16
evt code plen=4 status=disc 12 bits conn handle see table on page 766
has occurred 0x0001 reason for disc=conn term by local host

> HCI Event: Command Status(0x0f) plen 4 - Page 745
04 0F 04 00 08 00 00
evt code plen=4 comd curr pending num of packets command that is pending and caused evt
allowed to send to rok

```

## D.2.6.2 Server

```

L2PING (l2ping 11:11:11:50:11:11) - (11:11:11:50:11:11 get pinged by 00:00:00:00:00:04)
-----
here we are !

> HCI Event: Connect Request(0x04) plen 10 - Page 732
04 04 0A 04 00 00 00 00 00 01 00 01
evt Connection plen=10 6Bytes Address 0x000100 Link Type=ACL
Request 00:00:00:00:00:04 Class Of Device connection request

< HCI Command: Accept Connection Request(0x01|0x0009) plen 7 - Page 572
01 09 04 07 04 00 00 00 00 00 01
cmd ogf/ocf plen=7 6 Bytes Address Role=Slave
opcode

> HCI Event: Command Status(0x0f) plen 4 - Page 745
04 0F 04 00 08 09 04
evt command status plen=4 command num hci opcode that is pending
pending cmd packets

> HCI Event: Connect Complete(0x03) plen 11 - Page 730

```

```

04      03      0B      00      01 00      04 00 00 00 00 00      01      00
evt      Connection      plen=11      status=      2Bytes(12Bits meaningful) BD_ADDR      Link_Type=ACL      Encryption_Mode=
complete      success      0x0001=1      (Data Channel)      disabled

> ACL data: handle 0x0001 flags 0x02 dlen 28      -      Page 555
L2CAP(s): Echo req: dlen 20

02      01 20      1C 00      18 00 01 00 08      C8      14 00 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58
acl      handle/flags      dlen = 0x001c      the 28 acl data bytes
0x2001      = 28      L2CAP:
flags = 20 01 >> 12      18 00 Length 0x0018 = 24
= 0x02      see page 557      01 00 Signalling CID 0x0001 (all config req/resp L2CAP stuff has signalling cid 0x0001)
handle = 20 01 & 0fff      08 Code = Echo Request
= 0x0001      C8 Identifier = 200      (this is used to identify requests / responses)
14 00 Length 0x0014 = 20
45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 optional data...

< ACL data: handle 0x0001 flags 0x02 dlen 28      -      Page 555
L2CAP(s): Echo rsp: dlen 20

02      01 20      1C 00      18 00 01 00 09      C8      14 00 45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58
acl      as above      dlen      the 28 acl data bytes...
handle/flags      0x001c = 28      L2CAP:
18 00 Length 0x0018=24
01 00 Signalling CID 0x0001
09 Code Echo Response
C8 Identifier = 200
14 00 Length 0x0014 = 20
45 46 47 48 49 4A 4B 4C 4D 4E 4F 50 51 52 53 54 55 56 57 58 optional data

this goes on and on... always the correct identifier in
a Echo rsp acl packet has to be sent back !
pressing ctr+c in the l2ping console stops pinging...

> HCI Event: Disconn Complete(0x05) plen 4      -      Page 733

01      05      04      00      01 00      08 see table on page 766
cmd      disconn      plen=4      status=      2Bytes (12Bit Meaningful) Reason for disconnection=
complete      disconn has      connection handle      connection timeout
occured      0x0001

```

### D.2.7 l2test [9]

```

L2TEST sending data as a client to a server (l2test -s 00:00:00:00:00:04)
-----
< HCI Command: Create Connection(0x01|0x0005) plen 13      -      Page 568

01      05 04      0D      04 00 00 00 00 00      18 00      00      00      00 00 01
cmd      ogf/ocf      plen=13      bdaddr=      pkt type=      page scan      page scan      clock      allow_role_switch
opcode      00:00:00:00:00:04      0x0018      (acl)      rep mode R0      scan mode= mand      offset master may switch to slave

> HCI Event: Command Status(0x0f) plen 4      -      Page 745

```

```

04      0F      04      00      07      05 04
evt      code      plen=4 pending      num hci cmds allowed to pass to rok opcode of pending cmd

> HCI Event: Connect Complete(0x03) plen 11      -      Page 730

04      03      0B      00      01 00      04 00 00 00 00 00      01      00
evt      code      plen=11      status      12 bits      bdaddr      link type=ACL      encryption mode =disabled
success      connection

> HCI Event: Command Status(0x0f) plen 4      -      Page 745

04      0F      04      00      08      00 00
evt      code      plen=4 pending(=ok)      we can sent up to 8 cmds to rok      cmd that is pending (none)

< ACL data: handle 0x0001 flags 0x02 dlen 12      -      Page 555 / 286
L2CAP(s): Connect req: psm 10 scid 0x0040

02      01 20      0C 00      08 00 01 00 02      01      04 00 0A 00 40 00
acl      handle/flags      dlen      12 acl data bytes...      12cap cmd data (Page 286)
0x000c = 12      L2CAP:
08 00 0x0008 Length (of cmd)
01 00 0x0001 CID of all signalling cmds
02 0x02 Cmd code (Connection request)
01 0x01 Identifier (helps to match requests/replys see Page 284)
04 00 0x0004 length of data field in cmd
0A 00 0x000A 10 (PSM)
40 00 0x0040 64 (Source CID)

> ACL data: handle 0x0001 flags 0x02 dlen 16      -      Page 555 / 287
L2CAP(s): Connect rsp: dcid 0x0040 scid 0x0040 result 0 status 0

02      01 20      10 00      0C 00 01 00 03      01      08 00 40 00 40 00 00 00 00 00
acl      handle/flags      dlen      the 16 acl data bytes      12cap cmd data (Page 287)
0x0010 = 16      L2CAP:
0C 00 0x000C Length of cmd 12
01 00 0x0001 CID of all signalling cmds
03 0x03 Code (connection response)
01 0x01 Identifier (like a sequence number)
08 00 0x0008 Length
40 00 0x0040 Destination CID
40 00 0x0040 Source CID
00 00 0x0000 Result
00 00 0x0000 Status
00 00 0x0000 Status
-      Page 555 / 288

< ACL data: handle 0x0001 flags 0x02 dlen 12
L2CAP(s): Config req: dcid 0x0040 flags 0x0000 clen 0

02      01 20      0C 00      08 00 01 00 04      02      04 00 40 00 00 00
acl      handle/flags      dlen      12 acl data bytes...      12cap cmd data (Page 288)
0x0010 = 16      L2CAP:
08 00 0x0008 Length
01 00 0x0001 CID of all signalling cmds
04 0x04 Code (Configuration Request)
02 0x02 Identifier

```

```

04 00 0x0004 Length 4
40 00 0x0040 Dest CID
00 00 0x0000 Flags

> ACL data: handle 0x0001 flags 0x02 dlen 14 - Page 555 / 291
L2CAP(s): Config rsp: scid 0x0040 flags 0x0000 result 0 clen 0

02 01 20 0E 00 0A 00 01 00 05 02 06 00 40 00 00 00 00 00
acl handle/flags dlen 14 acl data bytes 6 data bytes (Page 291)
0x000e = 14 L2CAP:
0A 00 0x000A (length=10)
01 00 0x0001 as above
05 0x05 code (configure response)
02 0x02 Identifier
06 00 0x0006 length
40 00 0x0004 Source CID
00 00 0x0000 Flags
00 00 0x0000 Result

< ACL data: handle 0x0001 flags 0x02 dlen 14 - Page 555 / 281
L2CAP(d): cid 0x40 len 10 [psm 10]

02 01 20 0E 00 0A 00 40 00 25 00 00 00 0A 00 7F 7F 7F 7F
acl handle/flags dlen 14 acl data bytes
0x000E = 14 L2CAP:
0A 00 0x000A Length
40 00 0x0040 Channel ID (not signalling !!!)
25 00 00 00 0A 00 7F 7F 7F 7F 7F 7F is Info
^^

* the next data packets that get sent look all the same... but the count field
(first data byte is incremented each time -> pseudo sequence number)
like ...

02 01 20 0E 00 0A 00 40 00 26 00 00 00 0A 00 7F 7F 7F 7F
^^

* from time to time, the rok sends back a...

> HCI Event: Number of Completed Packets(0x13) plen 5 - Page 749

04 13 05 01 01 00 04 00
evt code plen Number of handles Conn Handle HC num of completed packets
number of Number of handles * 2 bytes 2 bytes * number of handles
completed handle=0x0001 (12 bits) so 0x004 = 4 packets have completed
packets

* saying, that 4 data packets have been flushed (=sent out)

* or say, if rok answers this twice... the host sends 8 packets afterwards...
you have to check that and compare always to the maximum number of acl
packets on can sent to the rok (read at the beginning - read_buffer_size...)

* pressing ctrl+c results then in a abort of the packets beeing sent..

< ACL data: handle 0x0001 flags 0x02 dlen 12 - Page 555 / 293
L2CAP(s): Disconn req: dcid 0x0040 scid 0x0040

02 01 20 0C 00 08 00 01 00 06 03 04 00 40 00 40 00

```

```

acl      as above      dlen      12 acl data bytes...
                0x000c = 12      L2CAP:
                                08 00  0x0008 Length 8 bytes data
                                01 00  0x0001 Signalling
                                06      0x06 Cmd code (Disconnection Request)
                                03      0x03 Identifier 3
                                04 00  0x0004 Length
                                40 00  0x0040 DCID
                                40 00  0x0040 SCID

> ACL data: handle 0x0001 flags 0x02 dlen 12      -      Page 555 / 294
  L2CAP(s): Disconn rsp: dcid 0x0040 scid 0x0040

02      01 20      0C 00      08 00      01 00      07      03      04 00      40 00      40 00
acl      as above dlen=12 12 acl data bytes
                L2CAP:
                                08 00  0x0008 Length 8
                                01 00  0x0001 Singalling cid
                                07      0x07 Cmd Code = Disconnection Response
                                03      0x03 Identifier
                                04 00  0x0040 Length = 4 bytes
                                40 00  0x0040 DCID
                                40 00  0x0040 SCID

< HCI Command: Disconnect(0x01|0x0006) plen 3      -      Page 571

01      06 04      03      01 00      13
cmd      Opcode      plen      Connection handle Reason see P 571 = Other End Terminated Connection Error Codes
          ogf/ocf          12 bits meaningful
                          out of 0x0001

> HCI Event: Command Status(0x0f) plen 4      -      Page 745

04      0F      04      00      07      06 04
evt      code      plen=4      command no of packets      opcode of command that
          currently      one can still      caused event and is pending
          pending      send to rok

> HCI Event: Disconn Complete(0x05) plen 4      -      Page 733

04      05      04      00      01 00      16
evt      code      plen      disc has connection      Reason See p 766 Table 6.1
          occurred      handle 12 bits      =Connection terminated by local host
                          out of (0x0001)

> HCI Event: Command Status(0x0f) plen 4      -      Page 745

04      0F      04      00      08      00 00
evt      code      plen      cmd pending      no of pkt      opcode dummy
          rok accepts

```



**E**

*Vortrag*

## Reconfigurable Bluetooth-Ethernet Bridge

Diplomarbeit WS 01/02

Michael Lerjen  
Christian Zbinden

Betreuer: Jan Beutel, Herbert Walder  
Professor: Lothar Thiele

## Inhalt

- Ethernet Bluetooth Bridge
  - Vision
  - Implementation
  - Resultate
- Rekonfiguration
  - JBits
  - Task - Transformation
  - Resultate

## Vision

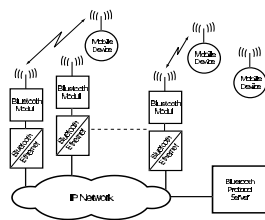
- Ganzes Gebäude mit Bluetooth-Infrastruktur ausgestattet
  - ⇒ Access-points (Gateway zu kabelbasierten Netzen)
  - ⇒ Positionsbestimmung
    - Navigation / Location based Services
    - Ortung (Personen / Ware)
  - ⇒ Multihop Verbindungen

## Bluetooth - Modul



- Modul: Management der Funkverbindung
- PC: Verbindungsmanagement, Anbindung an bestehende Protokolle

## Konzept

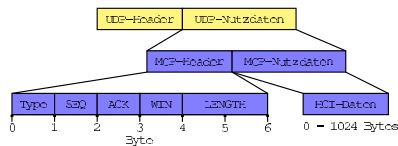


## Kommunikations-Protokoll

- Anforderung:
    - Fehlerfreie Verbindung
    - Verzögerung kleiner als ~1 sec (Timeouts)
  - Naheliegendste Lösung:
    - TCP
  - Probleme:
    - Paketierung nicht vorhersehbar
    - In HW sehr aufwendig zu realisieren
- ⇒ Eigenes Protokoll: MiniTCP

## Eigenes Protokoll - MiniTCP

### Header - Felder



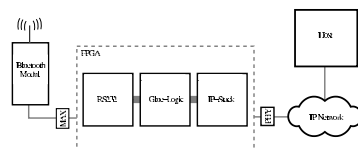
## Eigenes Protokoll - MiniTCP

### Hauptunterschiede zu TCP:

- Maximale Paketgröße: 1024 Bytes (vs 64k)
- Window-Größe in Paketen (vs Bytes)
- Sequenz-Nummern in Paketen (vs Bytes)
- Statische Timeouts

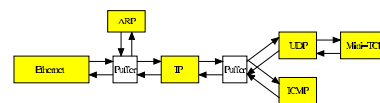
## Implementation (HW)

### Board mit Xilinx FPGA



## IP - Stack

### Blockdiagramm:



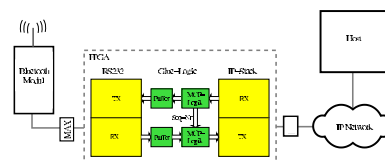
## Glue - Logic (1)

### Aufgaben:

- Verbindungsaufbau und -abbau
- Sequenznummern-Management
- Mehrere parallele Puffer für Rx und Tx

## Glue - Logic (2)

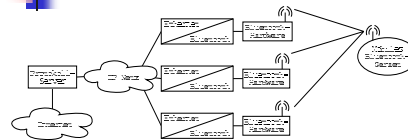
### Implementation:



### Implementation (SW)

- Erweiterung des vorhandenen Bluetooth-Stack (BlueZ) um eine Netzwerk-Schnittstelle

### Anwendungen



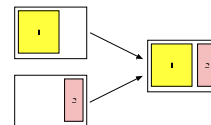
Siehe Demo (Do 10Uhr, G69)

### Resultate

- Prinzipielle Einschränkungen:
  - Timeouts
  - Sliding Windows begrenzt durch höhere Schichten
- Weiterführende Arbeiten:
  - Bridge: DHCP
  - Software: Service Discovery Protocol, Roaming, Handover

### Rekonfiguration

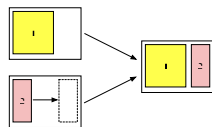
- Ziel:



Ziel: Tasks dynamisch ladbar (während Laufzeit)  
 ■ z.B.: Audio-Interface, USB-Interface

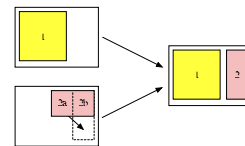
### JBits

- JBits als Werkzeug
  - Manipulation der FPGA-Konfiguration
  - FPGA-Strukturen als Java-Objekte
  - Verschieben und neu verdrahten



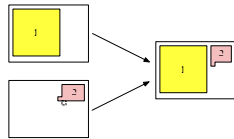
### Footprint-Transformation (1)

- Erhöht Flexibilität und Effizienz
  - Subtask-Level:



## Footprint-Transformation (2)

- Logikblock-Level
- Sehr flexibel
- aufwändig



## Resultate

- Footprint-Trafo prinzipiell möglich
  - Demonstration an kleinem Design
  - Routing-Probleme bei grösseren Designs
- Ungenügende Tools
  - Constraining des Routings müsste möglich sein
  - Intelligenterer Routing-Algorithmus nötig!



# Literaturverzeichnis

- [1] Bluetooth Special Interest Group: *Bluetooth V1.1 Core Specifications*. 2001  
<http://www.bluetooth.org/specifications.htm>
- [2] Bluetooth Special Interest Group: *Bluetooth V1.1 Profile Specifications*. 2001  
<http://www.bluetooth.org/specifications.htm>
- [3] Maksim Krasnyanskiy: *Bluez, Protocol Stack for Linux*  
<http://bluez.sourceforge.net>
- [4] Maksim Krasnyanskiy: *CVS-Tree of Bluez, Protocol Stack for Linux*  
<http://cvs.sourceforge.net/cgi-bin/viewcvs.cgi/bluez/>
- [5] Jan Beutel, Maksim Krasnyanskiy: *Linux Bluez Howto*, 2002  
<http://bluez.sourceforge.net/howto/index.html>
- [6] Käslin, Hubert: *VLSI I: Lecture Notes on Architectures of Very Large Scale Integration Circuits*. Microelectronics Design Center ETH Zürich, 2000
- [7] Riccardo Semadeni and Lars Wernli: *Bluetooth Unleashed, Wireless Netzwerke ohne Grenzen*. Semesterarbeit ETH Zürich, TIK, July 2001.
- [8] Jaap C. Haartsen: *The Bluetooth Radio System* IEEE Personal Communications February 2000
- [9] Egon Burgener, Peter Fercher: *Grenzenlose Piconetze mit Bluetooth*. Semesterarbeit ETH Zürich, 2002
- [10] Trolltech: *Trolltech Homepage*.  
<http://www.trolltech.com/products/qt/>
- [11] XESS *Homepage*.  
<http://www.xess.com>
- [12] Xilinx: *Virtex™ Data Sheet*.  
<http://www.xilinx.com/partinfo/ds003.htm>
- [13] University of Queensland, Brisbane: *VHDL XSV Board Interface Projects*.  
<http://www.itee.uq.edu.au/~peters/xsvboard>
- [14] Bandung Institute of Technology, Bandung: *10/100 Mbps Ethernet MAC*.  
<http://ic.ee.itb.ac.id/~hendrag97/ethmac.pdf>

- [15] Xilinx: *JBits SDK Version 2.8 for Virtex Documentation*.  
Xilinx, San Jose CA
- [16] Sun Microsystems: *Java Homepage*.  
<http://www.java.sun.com>
- [17] Xilinx: *Xilinx Architecture Description*.  
Xilinx, San Jose CA
- [18] Xilinx: *FAQ - Partial Reconfiguration*.  
<http://www.xilinx.com/xilinxonline/partreconfaq.htm>
- [19] Herbert Walder, Marco Platzner: *Non-preemptive Multitasking on FPGAs: Task Placement and Footprint Transform*.
- [20] Zippy Homepage: *A Novel Dynamically Reconfigurable Processor*.  
<http://www.tik.ee.ethz.ch/~zippy>
- [21] Michael Beck: *Linux-Kernelprogrammierung, Algorithmen und Strukturen der Version 2.4*, Addison-Wesley, 2001
- [22] Alessandro Rubini, Jonathan Corbet: *Linux Device Drivers*. 2nd Edition, O'Reilly, 2001  
<http://www.xml.com/ldd/chapter/book>
- [23] Glenn Herrin: *Linux IP Networking, A Guide to the Implementation and Modification of the Linux Protocol Stack*, 2000  
<http://kernelnewbies.org/documents/ipnetworking/linuxipnetworking.html>
- [24] Alan Cox: *Network Buffers and Memory Management*. Linux Journal, Nr 30 October 1996  
<http://www2.linuxjournal.com/lj-issues/issue30/1312.html>
- [25] Brian "Beej" Hall: *Beej's Guide to Network Programming, Using Internet Sockets*, 2001  
<http://www.ecst.csuchico.edu/beej/guide/net/>
- [26] Thamer Al-Herbish: *Raw IP Networking FAQ*. Version 1.3, 1999  
<http://www.whitefang.com/rin>
- [27] Mike D.Schiffman: *Libnet, Packet Assembly System, Version 1.0.2a*  
<http://www.packetfactory.net/Projects/Libnet/>
- [28] *Libpcap*, Packet Capture Library, Version 0.7.1  
<http://www.tcpdump.org/>
- [29] David S.Lawyer: *Serial HOWTO*. Version 2.14, August 2001  
<http://linuxdoc.org/HOWTO/Serial-HOWTO.html>
- [30] Gary Frerking: *Serial Programming HOWTO*. Version 1.01, 2001  
<http://www.linuxdoc.org/HOWTO/Serial-Programming-HOWTO/>

- [31] Matthias Dyer, Marco Wirz: *Build your own CPU on FPGA*.  
Diplomarbeit ETH Zürich, 2002
- [32] Gaisler Research: *LEON SPARC processor*:  
<http://www.gaisler.com/leon.html>
- [33] J. Postel: *RFC 768: User Datagram Protocol*. 1980  
<http://www.rfc.net/rfc768.html>
- [34] University of Southern California: *RFC 793: Transmission Control Protocol*.  
1981  
<http://www.rfc.net/rfc793.html>