

Semester Arbeit

Grenzenlose Piconetze mit Bluetooth

Wintersemester 01/02
7. Februar 2002

Burgener Egon
Fercher Peter

Betreuer: Jan Beutel, Prof. Dr. Lothar Thiele

Inhaltsverzeichnis

1	Vorwort	5
2	Hintergründe	7
2.1	Mobile ad hoc Netzwerke	7
2.2	RDSR - Routing in ad hoc Netzwerke	8
2.3	Spezifikation von xhop	9
2.4	Anwendungen	10
2.5	Bluetooth Grundlagen	11
3	BlueZ - Bluetooth Stack für Linux	13
3.1	Betrieb von BlueZ	13
3.2	Protokoll Schichtenmodell	14
3.2.1	Client	15
3.2.2	Server	16
4	xhop-Protokoll im Detail	17
4.1	RDSR Kopf	17
4.2	Kommando-Daten	18
4.3	Antwort-Daten	20
4.4	Benutzung von xhop	21
4.5	Implementation	22
5	Bluetooth auf AVR	25
5.1	Ausgangspunkt	25
5.1.1	Hardware	25
5.1.2	Software	25
5.2	Protokoll Stack	26
5.2.1	Software Komponenten	26
5.3	Schnittstellen	29
5.4	Protokoll Analyse	29
5.4.1	L2CAP	30
6	μxhop - xhop auf AVR	33
6.1	Software Tests	34
6.1.1	l2ping	34
6.1.2	xhop - ohne Kommando Daten	34
6.1.3	xhop - mit Kommando Daten	35
7	Fazit und Ausblick	37
7.1	Fazit	37
7.2	Ausblick	37
A	Protokoll Analyse	39

B Demo ping	53
C Demo xhop	57

1 Vorwort

Dieser Bericht betrifft unsere Semesterarbeit am Institut für Technische Informatik und Kommunikationsnetze (TIK) des Departements Informationstechnologie und Elektrotechnik der ETH Zürich. Unser Ziel ist es, hiermit nach einer kurzen Einführung in die verwendeten Technologien ein genaueres Verständnis für die entstandene Arbeit zu vermitteln. Dieser Bericht soll auch als Hilfe zur Verwendung unserer Implementation dienen. Zu diesem Zweck wende man sich an die Kapitel 3.2 4.5 6.

Unsere Semesterarbeit war sehr interessant und vor allem auch abwechslungsreich, da wir uns mit verschiedensten Technologien wie mobile ad hoc Netzwerke, Bluetooth und Mikrokontroller Programmierung beschäftigen mussten. Auf Grund dieser Vielfalt benötigten wir jedoch auch viel Zeit, uns immer wieder einzuarbeiten und fremden Code zu analysieren. Im grossen und ganzen sind wir gut vorangekommen und konnten unsere Ziele erreichen. Es war eine sehr lehrreiche Arbeit.

Bedanken möchten wir uns bei Jan Beutel, unserem Betreuer, der uns immer gerne und rasch geholfen hat und uns bei der Verrichtung unserer Arbeit viel Freiheit liess. Ebenso möchten wir uns bei der Dienstgruppe des Instituts bedanken, bei der wir immer umstandslos das notwendige Material beziehen konnten.

Burgener Egon und Fercher Peter

2 Hintergründe

In diesem Kapitel wollen wir einen kurzen Einblick in die Technologien und in die Problemstellung unserer Arbeit geben. Nähere Informationen zu mobilen ad hoc Netzwerken und Routing in ad hoc Netzwerken findet man im Bericht zur Semesterarbeit "Bluetooth Unleashed" [1]. Wer sich intensiver mit Bluetooth befassen will, lese in der Bluetooth Spezifikation [2] nach.

2.1 Mobile ad hoc Netzwerke

Mobile ad hoc Netzwerke sind dynamische Gebilde. In einem solchen Netzwerk sind die einzelnen Knoten nicht fix, sondern bewegen sich und dürfen das Netz zu beliebigen Zeiten verlassen bzw. sich zu beliebigen Zeiten anschliessen. Im Gegensatz zu statischen Netzwerken existiert beim ad hoc Netzwerk keine zentrale Steuer- und Routingstation. In der Abbildung 2.1 ist ein solches ad hoc Netzwerk dargestellt. Die Funktechnologie Bluetooth lässt sich für solche Netzwerke sehr gut verwenden. Die Funkreichweite

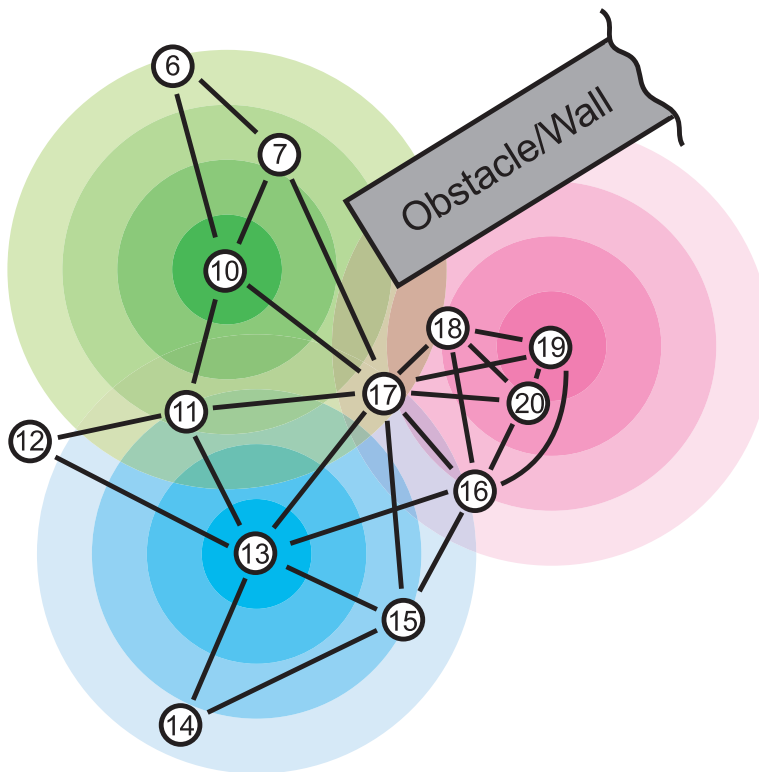


Abbildung 1: Mögliches ad hoc Netzwerk [3]

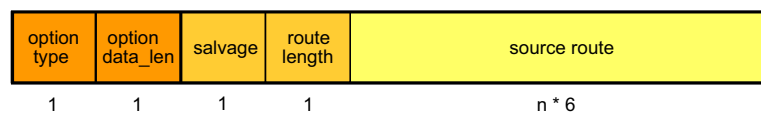
ist auf 10m beschränkt und somit für kleinere lokale Netze gedacht. Ein Bluetooth Knoten kann bis zu 7 Verbindungen zur gleichen Zeit halten (falls die Hardware dies unterstützt). Bei einem sogenannten Piconetz handelt es sich um ein Netzwerk bestehend aus maximal 8 Knoten wobei einer die Rolle des Masters übernimmt. Solche Piconetze, in der Abbildung 2.1 mit verschiedenen Farben dargestellt, können zu einem grossen Scatternetz zusammengeschaltet werden. Die Kommunikation zwischen zwei benachbarten Piconetzen geschieht mit Hilfe eines gemeinsamen Knotens, welcher in beiden Netzen Slave ist.

Zur Illustration nehmen wir an, dass der Knoten 14 mit Knoten 6 kommunizieren will. Da letzterer nicht im Funkradius von 14 liegt, müssen z.Bsp. Knoten 13, 11 und 10 die Pakete weiterleiten. Auf Grund der Dynamik der Netze und der grossen Anzahl Knoten, die somit verbunden werden können, sind komplexe Routingmechanismen notwendig.

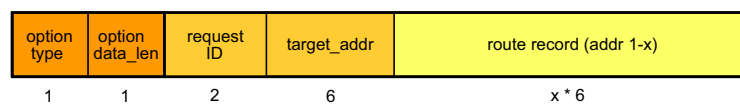
2.2 RDSR - Routing in ad hoc Netzwerke

Wie oben erwähnt, benötigt man in solchen ad hoc Netzwerken spezielle Routing Konzepte. Den Ergebnissen der Semesterarbeit "Bluetooth Unleashed" [1] zur Folge eignet sich das On-Demand Routing Protokoll Dynamic Source Routing (DSR) [9] am Besten. In der Arbeit wird ein Reduced Dynamic Source Routing (RDSR) vorgeschlagen. Folgende Paket Typen wurden spezifiziert:

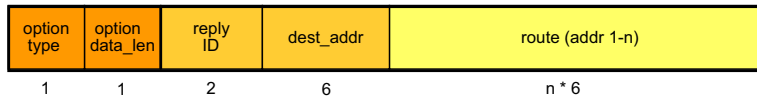
- Data



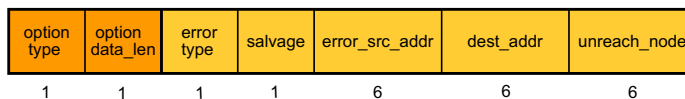
- Route Request



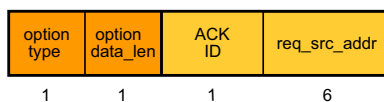
- Route Reply



- Route Error



- ACK Request



- ACK



Will nun ein Knoten Daten zu einem anderen Knoten senden, sucht er nach einer eventuell vorher abgespeicherten Route und sendet das "Data" Paket. Findet er jedoch keine Route, so sendet er ein "Route Request" Paket an alle seine Nachbarn, welche ebenso nach einer Route in ihrem Speicher suchen. Falls diese Suche misslingt, senden diese auch ein "Route Request" Paket. Dies geht solange weiter, bis der Endknoten erreicht wurde oder jemand eine Route gespeichert hat. In beiden Fällen wird die Route dem Initiator zurückgeschickt. Sobald der Sender die Route erhält, sendet er das "Data" Paket mit Hilfe der soeben erforschten Route.

2.3 Spezifikation von xhop

Unsere Aufgabe war es nun, einen Teil des RDSR Protokoll zu implementieren: Versenden, Weiterleiten und Empfangen von Data Paketen. Nach Aufgabenstellung wird die Kenntnis der Routen als bekannt vorausgesetzt.

Unsere Implementation des RDSR Protokolls trägt den Namen xhop. Das Paket Format von xhop wurde schon in der Semesterarbeit "Bluetooth Unleashed" [1] festgelegt und wir mussten uns soweit wie möglich daran halten.

Im Gegensatz zu unseren Vorgängern, welche ihre Arbeit auf dem Axis Bluetooth Stack [4] aufbauten, benutzten wir den stabileren und weiter entwickelten BlueZ Bluetooth Stack [5]. Mehr dazu im Kapitel 3.

Die Aufgabe von xhop ist, die Grenze von 10m Reichweite, sowie die Begrenzung von 8 Knoten pro Netzwerk zu überwinden. Jeder Knoten muss Verbindungen entgegennehmen können und falls er nicht das Endziel der Route ist, muss der Knoten das Paket weiterleiten. Falls der Knoten Endpunkt der Route ist, führt er die im Paket enthaltenen Kommandos aus. Da manche Kommandos Werte zurückliefern, muss diese Information an das Ende des Pakets angefügt werden, damit der Initiator diese Daten verarbeiten kann.

2.4 Anwendungen

Nach dieser kurzen Einführung in ad hoc Netzwerke und xhop stellt sich die Frage wozu man diese überhaupt gebrauchen kann.

Da mit Bluetooth verschiedene Arten von Daten übermittelt werden können, sind viele Anwendungen möglich. Hier möchten wir nur einige für uns relevante Anwendungen auflisten.

- **Positionsbestimmung**

Mit Hilfe der Bitfehlerrate und Signalleistung einer Verbindung kann die Distanz zum Kommunikationspartner ziemlich gut abgeschätzt werden. Durch mehrere Messungen und einer Mittelung kann die Genauigkeit weiter erhöht werden. Führt man nun diese Distanzmessung von mehreren Knoten aus, so kann durch Triangulation die Position im 3-dimensionalen Raum bestimmt werden. Solange kein Knoten fix ist, bleibt die gemessene Position relativ. Um absolute Werte zu erhalten, müssen mindestens 3 Knoten des Netzes fixiert und mit absoluten Werten versehen werden.

- **Anwendungen die auf die Positionsbestimmung aufbauen**

Auf die oben beschriebene Positionsbestimmung lassen sich nun weitere Anwendungen bauen. Es könnte ein Service entwickelt werden, mit dem man sich z.B. die Position der nächstgelegenen Pizzastube oder Pubs auf das Bluetooth Handy anzeigen lassen kann.

- **Dienstleistungsserver**

Wie in einem statischen Netz können Knoten mit Serverfunktionen

installiert werden. Diese können im Beispiel der Positionsbestimmung die genaue Position anderer Geräte übermitteln. Ein weiteres Beispiel wäre ein Server, der das Datum und die genaue Zeit zur Verfügung stellt.

- **Mobile Datenkommunikation**

Notebooks und auch normale Desktop Computer können im Nu miteinander vernetzt werden und dies ohne die Beschränkung durch die Reichweite und Anzahl Geräte. Ebenso können Peripheriegeräte ins Netz angebunden werden.

2.5 Bluetooth Grundlagen

Wie oben schon erwähnt, handelt es sich bei Bluetooth um einen Standard für die lokale Funkkommunikation. Es können sowohl via asynchrone wie auch über synchrone Verbindungen digitale und PCM Daten übermittelt werden (falls die Hardware dies unterstützt). Die Sendefrequenz liegt bei 2.4 GHz. Als Modulation wird die Bandspreiztechnik "frequency hopping" (FH) und Zeitmultiplexierung (TDM) angewendet. Die Wechselrate beträgt 1600 Wechsel pro Sekunde. Die gesamte verwendete Bandbreite beträgt 79 MHz. Tabelle 1 zeigt die möglichen Übertragungsraten. Es existieren zwei Übertragungstypen: Asynchron mit 721/56 Kbit pro Sekunde sowie synchron mit je 432 Kbit pro Sekunde.

Tabelle 1: Übertragungsraten

Asynchrone Übertragung	721/56	Kbit/s
Synchrone Übertragung	432	Kbit/s
Maximale Übertragungsrate	1	Mbit/s

Im Kapitel 2.1 "Mobile ad hoc Netzwerke" wurde schon beschrieben, dass ein Bluetooth Modul fähig ist, bis zu sieben Punkt-zu-Punkt Verbindungen gleichzeitig zu halten. Das Modul übernimmt die Rolle des Masters des so entstandenen Piconetzes. Die maximale Anzahl Slaves eines Piconetzes ist auf 256 Module beschränkt. Bei den Slaves unterscheidet man unter den aktiven Modulen, die eine bestehende Verbindung mit dem Master besitzen und den passiven, die keine Verbindung zum Master besitzen aber trotzdem zum Piconetz gehören. Alle Slaves des Netzes erhalten eine 8 Bit lange Page Identifikationsnummer. Die aktiven Slaves kriegen zusätzlich eine 3 Bit lange Identifikationsnummer. Alle passiven Slaves speichern sich Synchronisationsdaten wie Kanal Sequenz und Phase des Master und können dadurch schnell wieder in den aktiven Zustand wechseln. Dies ist für ein gut funktionierendes Netzwerk essentiell, wenn man bedenkt, dass eine Synchronisation mehrere Sekunden dauern kann.

Als nächstes betrachten wir die Protokoll Schichten, wie sie in Abbildung 2 dargestellt sind. Die Schichten unter der HCI Schnittstelle sind in Hardware realisiert. Das "Host Controller Interface" (HCI) ist die Schnittstelle zwischen dem Bluetooth Modul und dem Gerät welches das Modul steuern und benutzen will.

Auf der nächst höheren Schicht liegt das standardisierte "Link Layer Control and Adaptation Protocol" (L2CAP). Seine Aufgabe ist es, den darüber liegenden Schichten eine abstrakte, modulunabhängige Kommunikationsschnittstelle zu bieten. Es unterstützt durch multiplexen die Möglichkeit, mehrere logische Verbindungen über dieselbe physische Verbindung zu leiten. Dies geschieht über sogenannte "channel identifiers". Ebenso kontrolliert L2CAP die Fragmentierung und Defragmentierung der Daten. Auch "Quality of Service" (QoS) wird angeboten.

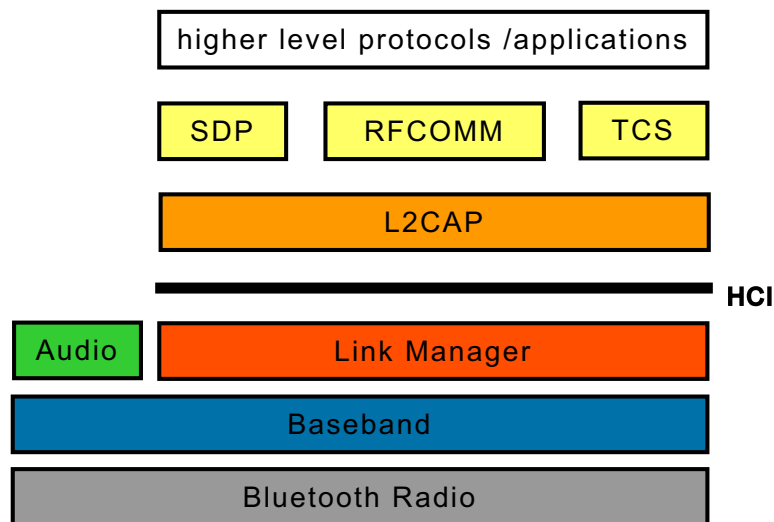


Abbildung 2: Bluetooth Protokoll Schichten [1]

3 BlueZ - Bluetooth Stack für Linux

Dieses Kapitel beschreibt den von uns verwendeten Bluetooth Stack BlueZ. BlueZ wird von der Open Source Gemeinde entwickelt und der Source Code ist bei sourceforge.net unter [5] erhältlich. Weil der Release 1.0 ab dem Kernel 2.4.10 nicht mehr kompatibel ist, benutzten wir die CVS Version vom 24 Okt. 2001. Seit Beginn des Jahres 2002 ist ein neuer Release 2.0-pre1 erhältlich.

3.1 Betrieb von BlueZ

Die Installation von BlueZ verläuft gemäss dem "How To" [6]. Dabei ist zu beachten, dass folgende Optionen im Kernel vorhanden sind:

- Networking Options \implies Kernel/User netlink socket.
- Network device support \implies PPP(point-to-point) support.

Weiter ist zu achten, dass "USB support \implies Bluetooth support" nicht gewählt ist.

Das BlueZ Paket enthält ein paar nützliche Hilfsprogramme wie `l2ping` und `l2test`, mit denen man die Installation und die Bluetooth Module testen kann. Nicht enthalten ist der "packet sniffer" `hcidump`, welcher jedoch auch unter [5] erhältlich ist. Startet man `hcidump` mit `[--hex|--raw]`, so wird jeweils auch das gesamte Paket in hexadezimaler Schreibweise dargestellt. Dies war für die Analyse der Funktionsweise und zum Suchen von Fehlern sehr hilfreich.

Mit dem Programm `hciconfig`, welches auch im Paket Bluez enthalten ist, kann das Bluetooth Modul konfiguriert und gesteuert werden. Zum Starten des Moduls verwendet man `hciconfig hciX up`, wobei X die interne, logische Nummer des Modules ist. Falls eine UART Verbindung zum Modul verwendet wird, muss man es zuerst an eine serielle Schnittstelle anbinden. Dazu kann entweder das Programm `hciattach` oder `hcid` verwendet werden. Wir haben folgende Syntax verwendet:

```
[root@localhost]# hciattach /dev/ttyS0 ericsson 57600 flow
```

Nähere Informationen zum Gebrauch findet man jeweils unter den Hilfe Optionen der einzelnen Programme.

Wie wir festgestellt hatten, läuft BlueZ stabil und zuverlässlich, falls eine UART Verbindung mit 57600 baud/s zum Bluetooth Modul verwendet wird. Leider hatten wir mit einer USB Verbindung Probleme, die jedoch

auch auf den Linux Kernel zurückzuführen sind. Zum Zeitpunkt unserer Arbeit wurde gerade an den USB Treibern Veränderungen vorgenommen, was zu Inkompatibilitätsproblemen führte. Weiter stellten wir fest, dass die Funktion `recv` der L2CAP Socket Schnittstelle manchmal 0 Bytes zurück gibt, obwohl das der Host das Paket vollständig erhalten hat. Dies konnten wir mit `hcidump` beobachten.

3.2 Protokoll Schichtenmodell

Abbildung 3 zeigt das Schichtenmodell des BlueZ Protokollstacks. Unter "BlueZ utilities" versteht man die kleinen oben beschriebenen Hilfsprogramme.

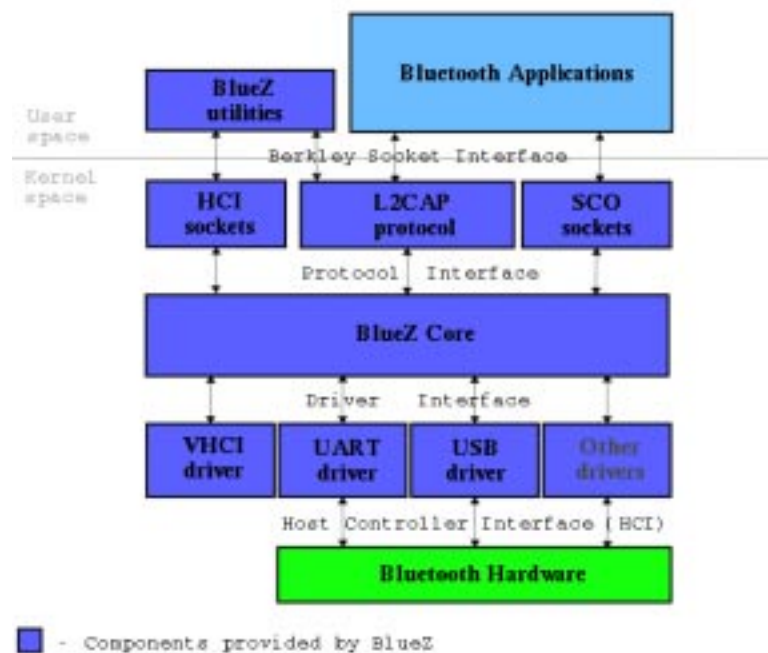


Abbildung 3: BlueZ Bluetooth Protokollstack

Die Schnittstelle zur L2CAP Schicht basiert auf "Sockets". Im folgenden sind die Schritte eines Clients sowie eines Servers skizziert. Den ausführlichen Code findet man in der Datei `l2test.c`.

3.2.1 Client

Unter Bluetooth ist der Begriff "initiator" gebräuchlicher. Will man Daten an einen Server senden, geht man folgendermassen vor: Zuerst holt man sich vom OS einen "socket filedescriptor".

```
s = socket(PF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
```

Jetzt wird dieser Socket an eine lokale Adresse gebunden. Es ist möglich, mehrere Bluetooth Devices am gleichen Computer angeschlossen zu haben. Mit `loc_addr` wählt man eines aus.

```
bind(s, (struct sockaddr *) &loc_addr, sizeof(loc_addr));
```

Optional kann man nach dem `bind` call mit

```
getsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, &opts_len);
```

Socket Optionen vom lokalen Socket lesen.

Neue Optionen setzt man mit:

```
setsockopt(s, SOL_L2CAP, L2CAP_OPTIONS, &opts, opts_len);
```

als Optionen können `opts.omtu`, und `opts.imtu` gesetzt werden. Dies sind die maximalen Transfereinheiten für Daten die über den Socket gesendet bzw. empfangen werden.

Nach `bind` kann man zur `rem_addr` verbinden der man Daten senden will:

```
connect(s, (struct sockaddr *)&rem_addr, sizeof(rem_addr));
```

Nach dem erfolgreichen `connect()` kann man nun Daten zum Server senden. Dazu benutzt man die Funktion:

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

Will man die Verbindung zum Server trennen, so benutzt man die Funktion:

```
close(s);
```

Hier sind wir auf ein weiteres Problem des BlueZ L2CAP Layers gestossen: Beobachtet man die Aktivität auf dem Socket mit `hcidump` so sieht man das der L2CAP Layer erst ca. 10 Sekunden nachdem `close()` zurückkehrt auch wirklich in den closed Zustand geht. Wir mussten darauf

achten, dass zwischen dem Abbruch einer Verbindung mit A und dem Aufbau einer Verbindung mit B dem L2CAP Layer genügend Zeit bleibt, um wieder in den Anfangszustand zu gehen. Unter Linux war deshalb eine Verzögerung von ca. 10 Sekunden notwendig (`sleep(10);`).

3.2.2 Server

Der Begriff "acceptor" ist gebräuchlicher. Will man Daten empfangen geht man folgendermassen vor: Zuerst öffnet man die Kommunikationsschnittstelle (man holt sich einen Socket):

```
s = socket(PF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
```

Danach wird der Socket an eine lokale Adresse gebunden:

```
bind(s, (struct sockaddr *) &loc_addr, sizeof(loc_addr));
```

Nun kann man auf dem Socket horchen, bis jemand eine Verbindung will. Dies geschieht mit:

```
listen(s, 10);
```

Will jemand verbinden, so kann man die Verbindung annehmen mit:

```
s1 = accept(s, (struct sockaddr *)&rem_addr, &opt);
```

Jetzt kann man mit `fork` einen neuen Prozess erzeugen, der sich um die Verbindung kümmert:

```
if( !fork() ) /* child */
```

Daten empfangen kann man mit der Funktion:

```
numbytes_received = read(s1, buf, data_size);
```

Will man die Verbindung trennen, so benutzt man:

```
close(s1);
```


4 xhop-Protokoll im Detail

In Kapitel 2 "Hintergründe" wurde die Funktionalität und Aufgabe von xhop beschrieben. Nun möchten wir in diesem Kapitel das Protokoll und dessen Verwendung im Detail besprechen.

Wie schon erwähnt, mussten wir uns beim Format von xhop an die Vorgaben der Semesterarbeit "Bluetooth Unleashed" [1] halten.

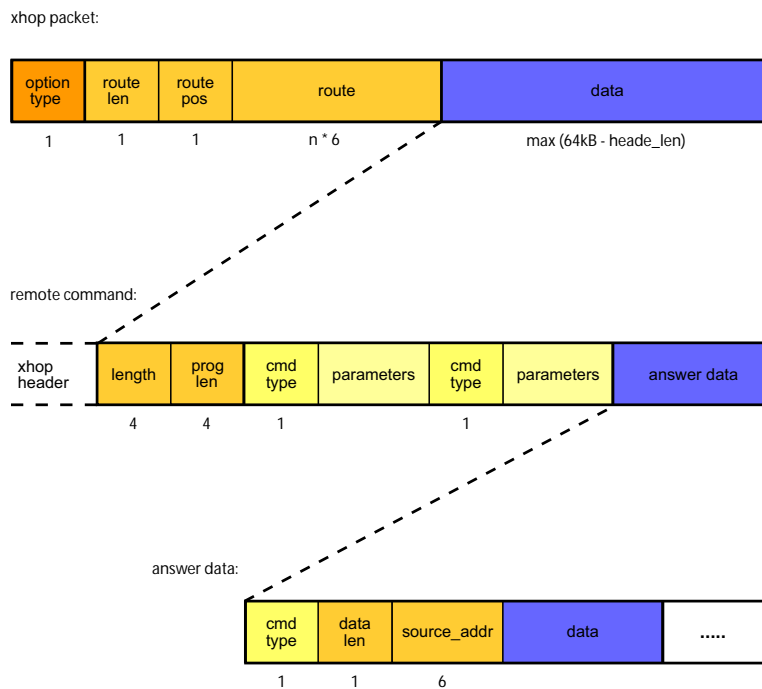


Abbildung 4: Paket Format von xhop [1]

Die Abbildung 4 zeigt ein Modell des Formats. Es ist gut erkennbar, dass das gesamte Paket in folgende drei Teile aufgeteilt ist:

1. RDSR Kopf
2. Kommando-Daten
3. Antwort-Daten

Im Folgenden werden diese einzelnen Teile genauer betrachtet.

4.1 RDSR Kopf

Das erste Byte des RDSR Kopfes beschreibt den Typ des RDSR Paketes. Dies ist für unsere Anwendung vom Typ RDSR Data. Beim zweiten Byte

handelt es sich um die Länge der Route, d.h. die Anzahl Knoten im Feld "Route". Das nächste Byte enthält die aktuelle Position in der Route. Der Wertebereich von 'route pos' beginnt mit 1 und endet mit dem Wert von 'route len'. Falls das Paket weitergeleitet werden muss, ist der Wert von "route pos" um eins zu inkrementieren.

Das Feld "route" enthält die 6 Byte grossen Bluetooth Adressen. Somit ergibt sich die Länge von "route" durch "route len" mal 6 Byte. Die Route muss vom Initiator des Pakets via RDSR oder sonstige Routingprotokolle berechnet worden sein. Es ist wichtig, dass jeder Knoten in der Route jeweils vom vorherigen direkt erreichbar ist.

4.2 Kommando-Daten

Beim zweiten Teil, den Kommando-Daten, handelt es sich um den Teil, der xhop für Anwendungen sehr nützlich macht. Die Idee ist, dass ein Bluetooth Knoten auf anderen Knoten Kommandos ausführen kann. Eines der möglichen Kommandos ist auch xhop selber. Somit ist man fähig mit nur einem xhop Paket zu einem entfernten Knoten zu "springen", dort Kommandos ausführen zu lassen und weiter zu einem beliebigen Knoten zu "springen", der ebenso Kommandos ausführt und das Paket weiterleiten kann. Die maximale Anzahl "Sprünge" ist lediglich durch die maximale Paketlänge von 64 KB begrenzt.

Der Kopf der Kommando-Daten enthält zwei Längenangaben. Die erste, als "length" bezeichnet, gibt die Gesamtlänge des Kommandoteiles an. Sie enthält die Länge des Kopfes von stets 8 Byte, die Länge der gesamten Kommandofelder und die Länge der gesamten Antwort-Daten.

Das Feld "prog len" enthält die Länge der gesamten Kommandofelder. Diese zweite Längenangabe dient dazu die Kommandofelder von den Antwort-Daten zu unterscheiden.

Jedes einzelne Kommandofeld besitzt einen vordefinierten Kopf mit 1 Byte langem Kommandotyp Feld und einem 4 Byte langen "param len" Feld. Dieses Parameterlängenfeld fügten wir hinzu, um das Durchlaufen des Paketes zu erleichtern. Abbildung 5 zeigt den für alle Kommandos geltenden Kopf.

Vorläufig haben wir folgende Kommando Typen implementiert:

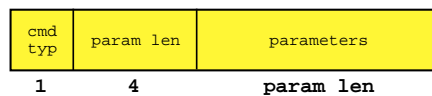


Abbildung 5: Kommando Kopf

- CMD_INQ
- CMD_SEND_DATA
- CMD_XHOP

Mit diesen drei Typen lassen sich viele Anwendungen aufbauen. Weitere Kommandos lassen sich ziemlich einfach einbauen. Dazu ist lediglich ein weiterer "case" Eintrag im "switch" der Funktion `exec_cmd` nötig.

Die Funktion von `CMD_INQ` ist, auf dem entsprechenden Knoten ein "inquiry" durchzuführen und die gefundenen Knoten den Antwort-Daten des Paketes hinzuzufügen. Bei einem "inquiry" sucht der Knoten nach Bluetooth Modulen die in seiner Reichweite sind. Bei kommerziellen Geräten wird diese Funktion oft auch als "discover" bezeichnet.

In Abbildung 6 ist das genaue Format vom Typ `CMD_INQ` dargestellt. Das Feld "sec" beinhaltet die Dauer der Suche in Sekunden. In der Regel sollten 7-8 Sekunden ausreichen um alle Bluetooth Geräte zu erkennen.

Mit dem Kommando `CMD_SEND_DATA` lassen sich beliebige Daten zu einem

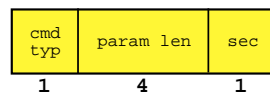


Abbildung 6: Kommando `CMD_INQ`

Knoten senden. In unserer Implementation benutzen wir dieses Kommando lediglich um einen Text am anderen Knoten anzeigen zu lassen. Das Feld "Data buffer" in Abbildung 7 enthält die Daten. Weiter ist zu beachten, dass die maximale Paketlänge eingehalten wird. Um eine grosse Flexibilität

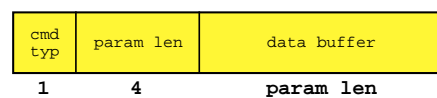


Abbildung 7: Kommando `CMD_SEND_DATA`

zu erreichen, benötigt man die Fähigkeit das Paket nach Ausführung von diversen Kommandos an weitere Knoten weiterzuleiten. Dies wird vorallem auch zum Rücksenden der Antwort-Daten verwendet.

Dieser Mechanismus wird mit dem Kommando `CMD_XHOP` erreicht. Wie Abbildung 8 zeigt, enthält das Kommando nur die Bluetooth Adresse des Bluetooth Moduls, zu dem das Paket gesendet werden soll. Der Knoten ist selber dafür zuständig eine Route zum Ziel zu finden. Da uns keine implementierten Routingprotokolle zur Verfügung standen, setzten wir das

Kommando so um, dass das Paket direkt zum Ziel versendet wird. Dies setzt die direkte Erreichbarkeit des Ziels voraus.

Bevor das Paket gesendet werden kann, wird der RDSR Kopf mit der neuen Route versehen. Weiter werden alle schon ausgeführten Kommandos inklusive `CMD_XHOP` von den Kommando-Daten entfernt.

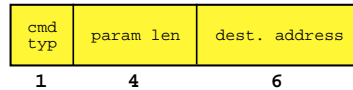


Abbildung 8: Kommando `CMD_XHOP`

4.3 Antwort-Daten

Wie wir oben gesehen haben, generieren manche Kommandos Rückgabewerte. Diese Daten werden ans Ende des Paketes angehängt. Damit diese später voneinander unterschieden und richtig zugeordnet werden können, ist ein gemeinsames Format, wie es in Abbildung 4 ersichtlich ist, nötig.

Das erste Byte "cmd typ" bestimmt von welchem Kommandotyp die Antwort stammt. So kann das Format der Daten abhängig vom Kommando entworfen werden. Das Feld "data len" enthält die Länge des "data" Feldes, das die eigentlichen Rückgabewerte beinhaltet, in Bytes. Das "source_addr" Feld gibt an von welchem Knoten die Daten stammen, d.h. es enthält die Bluetooth Adresse des Moduls, welches das Kommando ausgeführt hat.

Die Antwort-Daten werden vom Knoten genau dann gelesen, wenn keine Kommandos im Paket sind oder wenn man bei der Abarbeitung der einzelnen Kommandos nicht auf ein `CMD_XHOP` stößt.

Bei den drei von uns implementierten Kommandos, generiert lediglich `CMD_INQ` Rückgabewerte. Im folgenden wird das Format der Antwort-Daten eines `CMD_INQ` erläutert.

Das erste Byte im "data" Feld gibt die der Anzahl gefundenen Knoten an. Ist der Wert auf null gesetzt, so trat bei der Ausführung des "inquiry" ein Fehler auf oder es wurden keine Bluetooth Knoten gefunden. Im Falle eines Fehlers enthalten die darauffolgenden Bytes eine Fehlermeldung. War das "inquiry" jedoch erfolgreich, so sind in den restlichen Bytes die Bluetooth Adressen der gefundenen Module sequentiell aufgelistet.

4.4 Benutzung von xhop

In diesem Abschnitt zeigen wir, wie man xhop benutzt und wie man ein xhop Paket generiert. Da xhop nicht direkt vom Benutzer sondern von Anwendungen oder von höheren Schichten genutzt wird, haben wir weder eine graphische noch eine Konsolen basierte Schnittstelle geschaffen.

Das Programm xhop kann auf zwei verschiedene Arten gestartet werden. Zum einem im Empfangs-Modus mit dem Kommando `xhop -r` und zum anderen im Sende-Modus mit `xhop -s`. Im Empfangs-Modus führt das Modul ein "listen" aus und wartet bis eine Verbindung zustande kommt. Ist dies der Fall wird das Paket verarbeitet. Während der Verarbeitung bleibt das Modul beim "listen". Im Sende-Modus sendet das Modul das im Code definierte Paket. Die Hardcodierung des Pakets ist sehr simpel und nur für Test- und Entwicklungszwecke gedacht.

Der folgende Code Ausschnitt zeigt die Generierung des Pakets, welcher in der Funktion `demo_packet` in der Datei `xhop.c` zu finden ist.

```

...

//----- R O U T E -----
// ex.:
add_hop(buf, strtoba("00:00:00:00:00:44")); // Knoten A
add_hop(buf, strtoba("00:d0:b7:03:20:12")); // Knoten B
add_hop(buf, strtoba("00:00:00:00:00:07")); // Knoten C
//-----

...

//----- C O M M A N D S-----
// ex.:
// INQ(5);
// XHOP("00:00:00:00:00:07");

SEND(text,sizeof(text));
INQ(5);
XHOP("00:00:00:00:00:44"); // Knoten A (liesst die Antwort)
//-----

...

```

Als erstes muss die Route festgelegt werden. Dazu kann die Hilfsfunktion `add_hop` verwendet werden. Sie fügt die angegebene Bluetooth Adresse dem Feld "route" hinzu und inkrementiert "route len".

Als nächstes kann das Paket mit Kommandos versehen werden. Bei den Funktionen `INQ(int sec)`, `SEND(char *text,int len)`, `XHOP(char* baddr)` handelt es sich um Makros, die die Kommandofelder entsprechend generieren. Bei dem Argument von `XHOP` handelt es sich um eine Bluetooth Adresse, die mit 6 Hexadezimalzahlen getrennt durch Doppelpunkte geschrieben wird.

Möchte man eine Anwendung oder eine höhere Schicht aufbauend auf `xhop` entwickeln, so muss noch eine geeignete Schnittstelle erstellt werden.

4.5 Implementation

Wer sich mit dem Code von `xhop` beschäftigt, dem sei hier ein kleiner Überblick über die Implementation gegeben. Weitere Hilfe und Beschreibung der Funktionen steht im Code als Kommentar.

Wird `xhop` im Sende-Modus gestartet, so wird als erstes das Paket generiert und danach wird es dem zweiten Knoten in der Route gesendet. Das Senden übernimmt die Funktion `forward(char *buf, int len)`. Der Empfangs-Modus ist ein bisschen komplexer. Nach dem Start gelangt das Programm in die Funktion `do_listen` wo es wartet bis eine Verbindung zustande kommt. Ist dies der Fall so wird ein Kindprozess erzeugt, der die Verarbeitung des Pakets übernimmt. Der Vaterprozess bleibt weiter beim "listen" und wartet auf weitere Verbindungen. Abbildung 9 zeigt das Zustandsdiagramm des Empfangs-Modus.

Beim Kindprozess wird als erstes die Funktion `recv_handler` aufgerufen. Sie nimmt die empfangenen Bytes auf und speichert diese in einem Buffer.

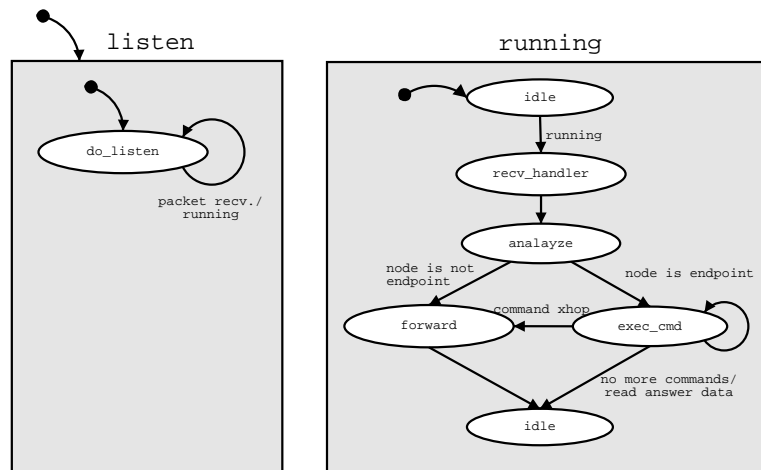


Abbildung 9: Zustandsmaschine von `xhop`

Als nächstes wird der Buffer der Funktion `analyze` übergeben, die das erhaltene Paket analysiert. Im Fall dass der Knoten nicht das Endziel ist, wird das Paket der Funktion `forward` übergeben, die das Feld "route pos" inkrementiert und das gesamte Paket an den nächsten Knoten weiterleitet und den Prozess terminiert. Ist der Knoten Endpunkt der Route, so müssen die Kommandos ausgeführt werden.

Die Analyse und Ausführung der Kommandos ist die Aufgabe der Funktion `exec_cmd`. Sind keine Kommandos vorhanden oder besitzt das Paket kein `CMD_XHOP` so werden die Antwort-Daten ausgelesen und der Prozess terminiert.

Im Falle, dass ein `CMD_XHOP` auftritt, wird ein neues Paket generiert. Das neue Paket enthält die neue Route, die Kommandos die noch nicht ausgeführt wurden sowie die bereits angehängten Antwort-Daten. Dieses neue Paket wird danach der Funktion `forward` zum Versand übergeben.

5 Bluetooth auf AVR

In diesem Abschnitt geht es um die Portierung auf den AVR. Ziel war es, die Funktionalität wie `l2test`, `l2ping` zu unterstützen. Also Daten zu senden und zu empfangen. Zudem wurde noch eine reduzierte Variante von `xhop` implementiert - siehe dazu auch das Kapitel 6.

5.1 Ausgangspunkt

5.1.1 Hardware

- **ATMEL 103 L**

Zur Verfügung stand uns der ATMEL 103 L. Der AVR hat eine Harvard Architektur mit separatem Speicher und Buss für Programmcode und Daten. Der Datenspeicher ist ein 4 KB grosses 8 Bit breites internes SRAM. Der Programmspeicher ist ein 64 KB grosses 16 Bit breites internes Flash. Der limitierte Speicher auf dem AVR reichte aber für unsere Implementation aus. Da der Datenspeicher jeweils byteweise adressiert wird, sollte man wenn möglich `_u8` anstelle von `int` benutzen. Dies lohnt sich speziell für "loopcounter", da der AVR für eine 32 Bit breite Variable 4 Ladeinstruktionen aus dem SRAM ausführen muss!

- **Bluetooth Modul**

Zur Verfügung stand uns der ROK 101 007 von Ericsson (Revision PA4).

5.1.2 Software

- **avr-gnutools**

Unsere Version der `avr-gnutools` haben wir von Oliver Kasten (D-INFK). An dieser Stelle auch unser Dankeschön an ihn, der uns die Diplomarbeit [7] erläutert hat. Die Tools können von seiner Webseite heruntergeladen werden:

<http://www.inf.ethz.ch/~kasten/research/bathtub/avr/>

Ebenfalls findet man dort Installationshinweise und den Link zu den neusten Versionen der Tools auf dem Web. Hilfreich kann auch das Abonnieren der Mailingliste für die `avr-gnutools` sein.

- **Diplomarbeit "The Event Collector Concept"**

Von unserem Betreuer wurde uns der Code der Diplomarbeit [7] zur Verfügung gestellt. Den Teil, den wir übernommen haben, stellt die Initialisierung des AVR sowie die Ansteuerung der seriellen Schnittstellen zur Verfügung.

- **smart-its**

Im Rahmen des Projektes smart-its [8], dessen Ziel es ist, Alltagsgegenstände durch kleine Computer zu verbinden, wurden wir auf die Arbeit von Frank Siegemund (D-INFK) aufmerksam gemacht. Er erklärte uns seinen Code und stellte ihn uns auch zur Verfügung. Es handelte sich um eine teilweise Portierung des Axis Protokoll Stacks. Auch ihm an dieser Stelle herzlichen Dank. Leider fehlte dieser Implementation die L2CAP Schicht, was den Grossteil unserer Arbeit ausmachte.

5.2 Protokoll Stack

Nach dem Studium dieser Voraussetzungen machten wir uns an die Implementation. Ziel war es, ein Programm zu schreiben, das es dem AVR ermöglicht mit einem Linux Rechner zu kommunizieren.

5.2.1 Software Komponenten

- **Makefiles**

Wir haben ein paar Makefiles geschrieben, die den Umgang mit den Tools erleichtern. Alle Änderungen werden im Toplevel Makefile vorgenommen (`Makefile.include`). Hier werden die wichtigsten Optionen und Pfadangaben gesetzt. Dieses Makefile kann dann bequem "included" werden. Als Beispiel betrachtet man am besten das Makefile zu `uxhop.c`. Mit

```
[root@pc-3294 uxhop]# make stk
```

wird der code für das Development Board übersetzt. Mit

```
[root@pc-3294 uxhop]# make btnode
```

wird der code für den btnode übersetzt. Mit

```
[root@pc-3294 uxhop]# make upload
```

wird der Code ins Flash vom AVR geschrieben.

- **h File**

Das File `hci_avr.h` ist eine Modifikation des BlueZ Stacks. Es enthält die nötigen `hci` und `l2cap` `#defines`

- **Debugging**

Debugging ist auf dem AVR ein wenig umständlich. Es geschieht über

die serielle Schnittstelle. Dazu wird am besten folgendermassen vorgegangen

- Linux Rechner mit AVR verbinden.
Dazu nimmt man entweder ein gerades 9-Poliges D-SUB (also 2 → 2, 3 → 3, 7 → 7, 8 → 8) oder wie wir, ein gekreuztes Kabel (also 2 → 3, 3 → 2, 7 → 8, 8 → 7) und schaltet einen Nullmodem Adapter dazwischen. (Die entsprechenden Kabel wurden von uns gelötet ;-)
- `minicom` auf Linux Rechner starten.
Die Optionen muss man auf 57600, 8N1 setzen. Danach sollte der Output vom AVR auf dem Linux Rechner in der `minicom` console sichtbar sein.
- `c` File
Im oberen Teil des Files `uxhop.c` stehen ein paar Zeilen, mit denen man die Debug Ausgaben des Programmes bequem steuern kann. Man muss dazu einfach die entsprechenden Zeilen auskommentieren. Beispiel:

```
/* enables debugging prints in send mode */  
#define __INITIATOR__
```

Dies ermöglicht compile-zeit gesteuertes Debuggen einzelner Komponenten.

Das Makro, das am meisten benutzt wird, ist

```
bprintf(s, arg...);
```

`s` ist ein "character array", ansonsten benutzt man die Funktion wie `printf`. Also zum Beispiel:

```
bprintf(my_buffer, "val=0x%x, cnt=%i \n \r", val, cnt);
```

- **`sprintf(s, arg...);`**

Für diese Funktion gibt es keinen Standard auf dem AVR. Deshalb haben wir eine Version, die auf dem Web unter

<http://home.arcor.de/0xdeadbeef>

zu finden ist, in unseren Code übernommen. Diese Funktion hat sich als robust und einfach erwiesen. Eine kurze Erklärung der Flags findet sich im File `uxhop.c`

- **Initialisierung des AVR**

Dies haben wir von der alten Diplomarbeit [7] übernommen.

- **Initialisierung des Bluetooth Modules**

Die Funktion `hci_dev_init()` übernimmt die Initialisierung und speichert die vom Modul erhaltenen Daten in einer Struktur, die auch ausgegeben werden kann. Das entspricht der Funktionalität von `hciconfig hcix up`.

- **Paket Parser**

Der Paket Parser ist in der Funktion `hci_recv_data()` implementiert. Seine Aufgabe ist es, Paketgrenzen zu erkennen. Mehrere Fälle mussten in Betracht gezogen werden:

- Paket kommt nur teilweise an.
- Paket wird ganz empfangen.
- mehr als ein Paket wird auf einmal empfangen.

Sobald ein Paket komplett empfangen wurde, wird der entsprechende Handler aufgerufen. Der Funktionsaufruf lautet:

```
hci_recv_data(acl_handler, evt_handler);
```

Eine Hilfsfunktion ist der `wrapper()`, der in einer "while loop" immer wieder `hci_recv_data()` aufruft. Es ist zu beachten, dass diese Art von Empfangen "blocking" ist. Falls "non-blocking" erforderlich ist, muss der "wrapper" angepasst werden.

- **Zustandsmaschine für Empfangen von Daten (`l2test -r`)**

Will man Daten empfangen, d.h. "listen mode", geht man folgendermassen vor:

```
wrapper (process_acl, process_evt);
```

Die Funktionen `process_acl()` und `process_evt()` bilden zusammen die Zustandsmaschine für den Empfangsmodus. Dieser Zustand des AVR ist identisch mit dem Linux Tool `l2test -r`. In diesem Modus kann man mit `l2test` Daten auf den AVR schicken. Dies kann wie folgt getestet werden:

```
[root@pc-3294 tools]# ./l2test -s 11:11:11:50:11:11
```

- **Zustandsmaschine für Senden von Daten (`l2test -s`)**

Will man aktiv Daten an eine Adresse senden, so benutzt man die Funktion

```
send_data(data_to_send);
```

Diese Funktion ruft wiederum `wrapper()` mit dem "handler" `l2cap_initiator()` auf. Dort ist die Zustandsmaschine für das Senden von Daten implementiert. Ob die Daten auch ankommen, kann man mit

```
[root@pc-3294 tools]# ./l2test -r
```

auf einem Linux Rechner überprüfen. Hilfreich ist es, gleichzeitig `hcidump` zu starten.

```
[root@pc-3294 tools]# hcidump --hex
```

5.3 Schnittstellen

Wie bereits im vorherigen Abschnitt beschrieben, existieren folgende Schnittstellen:

- Empfangen
`wrapper(process_acl, process_evt);`
- Senden
`send_data(the data to be sent...);`
- xhop
Die Funktion `uxhop()` wird nach dem erfolgreichen Empfangen von Daten aufgerufen. Hier werden die Daten analysiert und dann weitergeleitet.

5.4 Protokoll Analyse

Unser AVR-Port kann mit dem BlueZ Stack unter Linux kommunizieren. Für die Implementationen mussten wir herausfinden, welche Pakete in welcher Reihenfolge gesendet/empfangen werden. Deshalb wurden die folgenden Linux Kommandos aufgeschlüsselt:

- `hciconfig hci0 up`
- `hciconfig hci0 inq`
- `l2ping`
- `l2test`

Diese Tabellen befinden sich im Anhang A. Sie haben sich als sehr nützlich für die Implementation erwiesen.

5.4.1 L2CAP

- **L2CAP Verbindungsaufbau**

Der Verbindungsaufbau wird in der Abbildung 10 verdeutlicht. Als erstes sendet der Initiator ein `l2cap_conn_req`, welches vom Empfänger mit einem `l2cap_conn_rsp` beantwortet wird. Auf gleicher Art und Weise folgen `l2cap_conf_req` und `l2cap_conf_rsp`. Daraufhin schickt noch der Empfänger sein `l2cap_conf_req`. Dies beantwortet der Initiator wiederum mit einem `l2cap_conf_rsp`. Diese Prozedur dient dem Austausch von Verbindungsgrößen, wie Paketgröße, Danach kann der Initiator Daten zum Empfänger schicken.

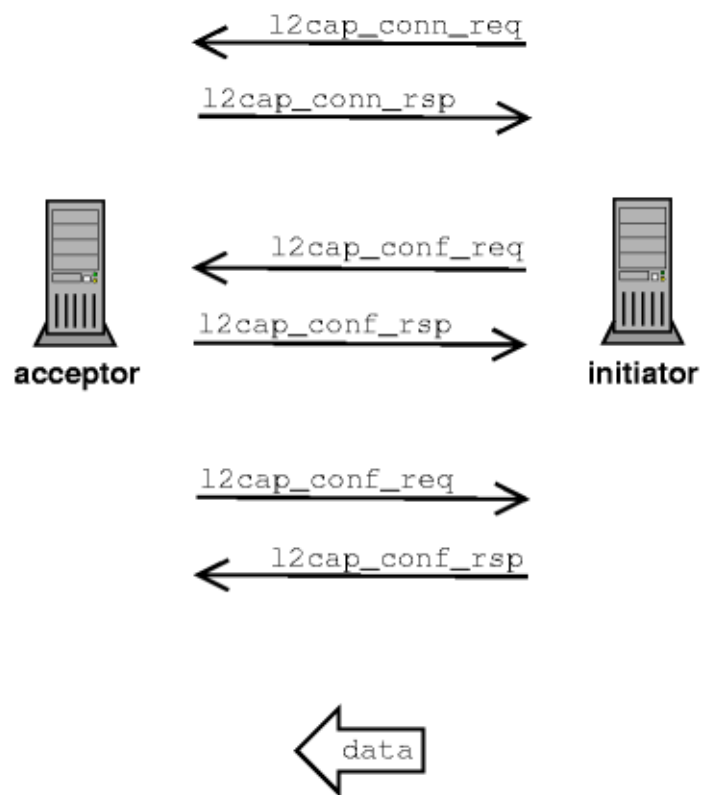


Abbildung 10: L2CAP Verbindungsaufbau

- **L2CAP Verbindungsabbruch**

Man betrachte die Abbildung 11. Nachdem der Initiator seine Daten gesendet hat, sendet er dem Empfänger ein `l2cap_disconn_req`. Der Empfänger antwortet mit einem `l2cap_disconn_rsp`, womit der Verbindungsabbruch beendet ist.

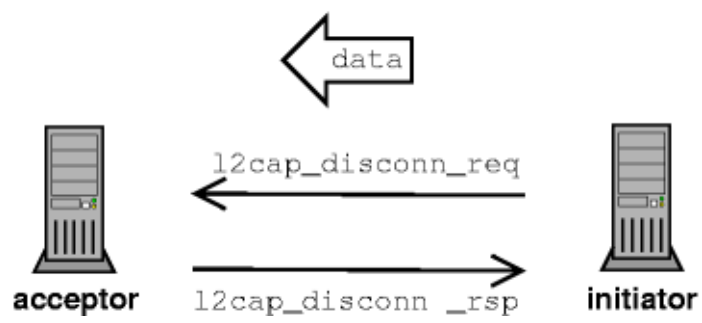
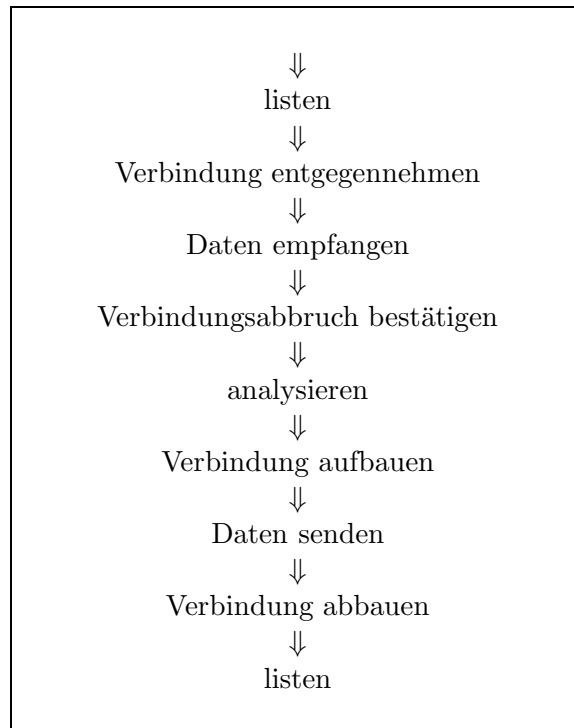


Abbildung 11: L2CAP Verbindungsabbruch

6 μ xhop - xhop auf AVR

Die Funktion `uxhop()` wird aufgerufen, sobald die gesendeten Daten auf dem AVR angekommen sind. `uxhop()` analysiert das Paket und leitet es an die entsprechende Adresse weiter.



Hier sei nun der Ablauf skizziert, die genaue Syntax findet man im File `uxhop.c`. Zuerst wird das angehängte Bluetooth Modul initialisiert. Dies entspricht dem Vorgang von `hcciconfig hci0 up`. Dabei wird die Information des Bluetooth Moduls in der Struktur `dev` gespeichert. Diese Struktur kann anschliessend ausgelesen werden.

```
hci_dev_init(&dev);
```

Dann kann ein Inquiry ausgeführt werden (Bsp.: 10 Sekunden, Maximal 200 Antworten):

```
inq(10, 0xc8);
```

Nun geht das Programm in den Empfangsmodus (acceptor) über:

```
wrapper(process_acl, process_evt);
```

Sobald Daten empfangen wurden, und die Verbindung wieder getrennt wurde, wird die Funktion `uxhop()` aufgerufen. Hier können die eingetroffenen Daten analysiert werden. Momentan werden die empfangenen Daten nur weitergeleitet:

```
void uxhop() {
    ...
    /* inc route pos field */
    h->route_pos++;
    /* send the data off */
    send_data(forward);
}
```

die Funktion `send_data` schickt Daten weiter an die Adresse `forward`

Nachdem die Daten versendet wurden, wird wieder auf eingehende Verbindungen gehorcht.

6.1 Software Tests

Die Entwicklung wurde fortlaufend mit Tools wie `l2test` oder `l2ping` `hcidump --hex` oder `hcidump --raw` geprüft. Die Funktionalität von `xhop` haben wir mit den zwei folgenden Szenarien überprüft.

- Knoten A, Linux Rechner
- Knoten B, AVR
- Knoten C, Linux Rechner

6.1.1 l2ping

Um die "ping" Funktionalität zu testen, nutzten wir das Programm `l2ping`. Das Protokoll findet man im Anhang B

6.1.2 xhop - ohne Kommando Daten

- Route
(A \rightarrow B \rightarrow C)

A sendet ein `xhop` Paket an B. B leitet das Paket nach C weiter. C ist der Endpunkt und überprüft, ob er Kommandos ausführen soll. Dem ist in diesem Fall nicht so. Schliesslich gibt C das Paket auf der Konsole aus. Das Protokoll dieses Tests befindet sich im Anhang C.

6.1.3 xhop - mit Kommando Daten

- Route
(A → B → C inq, xhop → B → A)

Auch das folgende Szenario konnte erfolgreich getestet werden:

Host A sendet ein xhop Paket an B. B forwardet das Paket an C. Dieser ist Endpunkt und führt nun die Kommandos aus, die ihm A aufgetragen hat. In diesem Fall ein `inquiry` und ein `xhop`. Danach werden die `inquiry` Daten von C ans `xhop` Paket angehängt und nach der von A vorgegebenen Route wieder über B nach A geschickt.

7 Fazit und Ausblick

7.1 Fazit

Mit der in dieser Semesterarbeit programmierten Software ist es möglich, ein Netz ohne harte Grenzen aufzubauen. Mit xhop ist es möglich auf einem entfernten Knoten ein Kommando auszuführen. Dies war ein Hauptziel unserer Aufgabenstellung. Mit der Portierung auf den AVR Mikrocontroller ist nun auch die Benutzung von xhop mit mobilen Knoten möglich.

Die Funktionalität unserer Programme wurde fortlaufend durch Tests geprüft.

7.2 Ausblick

Die weitere Arbeit besteht nun darin, eine funktionsfähige routing Software zu entwickeln, die mittels unseres Codes eine Topologie des Netzwerkes aufstellt. Eine graphisches Schnittstelle wäre sicher zu Testzwecken auch interessant. Weiter würde die Implementation von zusätzlichen Kommandos das Anwendungsgebiet von xhop verbreitern.

Ein nächster Schritt wäre die Entwicklung von Anwendungen die verschiedenste Dienste anbieten. Um diese Dienste zu finden, wäre noch "Service Discovery" Software nötig.

A Protokoll Analyse

Die nachfolgenden Seiten zeigen die byteweise Analyse der Programme:

- `hciconfig`
- `inquiry`
- `l2ping`
- `l2test`

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 - Page 744 / 702 / 243
04 0E 0C 08 03 10 00 00 00 00 00 00 00 00
evt code plen num hci cmd pkts cmd opcode status 00 = success FF F9 01 00 00 00 00 00
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 00 00 00 00 00 00
evt code plen num hci cmd pkts cmd opcode status acl max len num sco 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 00 00 00 00 00
evt code plen num hci cmd pkts cmd opcode status 55 00 00 00 00 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
[root@pc-3294 /root]# hciconfig hci0 down
< HCI Command: Reset(0x03|0x0003) plen 0 - Page 622
01 03 0C 00
cmd opcode plen
```


INQUIRY (hciconfig hci0 inq 5)

```

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

> HCI Event: Command Status(0x0f) plen 4          Page: 745
04 0F 04 00 00 04 01 04
evt  command plen 04 command opcode that is pending
      status      currently pending
      is currently pending

> HCI Event: Inquiry Result(0x02) plen 15         Page: 728
04 02 0F 01 Num_Resp=1
evt  inq  plen=15 Num_Resp=1
      result
      00:00:00:00:00:04
      6bytes * Num_Resp
      00: 1 Byte * Num_Resp: Page_Scan_Repetition_Mode (=R1)
      00: 1 Byte * Num_Resp: Page_Scan_Period_Mode (=P0)
      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
evt  inquiry complete plen=1
      success

> HCI Event: Command Status(0x0f) plen 4          Page: 745
04 0F 04 00 00 04 01 04
evt  command status plen=4 command
      currently pending      currently pending
      number of hci packets
      that one can still
      send to rok
      Opcode of command
      which caused this event and is pending completion
  
```


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

AAAAAAAAAAAAAAAA
here we are !

- Page 568

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

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

- Page 745

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

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

- Page 730

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

04 03 0B 00 01 00 04 00 00 00 00 01 00
evt Connection Complete Connection Handle(12 Bits) address= Link_Type= Encryption_Mode=
success 00:00:00:00:00:04 ACL disabled

- Page 745

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

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

- Page 555 / 293

< ACL data: handle 0x0001 flags 0x02 dlen 28

L2CAP(s): Echo req: dlen 20
02 01 20 1C 00 18 00 01 00 08 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 bytes
01 00 0x0001 Cid (signalling) 08 0x08 L2cap cmd code (Echo request)
C8 0xc8 Identifier = 200
14 00 0x0014 Length of cmd

- Page 555 / 294

> ACL data: handle 0x0001 flags 0x02 dlen 28

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

- Page 571

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

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

- Page 745

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

04 0F 04 00 status=command pending opcode of pending cmd 4 cmds can currently be sent to rok
07 06

- Page 733

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

04 05 04 00 01 00 16
evt code plen=4 status=disc has occurred 12 bits conn handle see table on page 766
0x0001 reason for disconn term by local host

- Page 745

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

04 0F 04 00 status=command pending num of packets allowed to send to rok
08 command that is pending and caused evt
00 00

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

^^^^^^^^^^^^^^^^^^^^^^
here we are !

```
> HCI Event: Connect Request(0x04) plen 10 - Page 732
04 04 0A 04 00 00 00 00 00 01 00
evt Connection Request plen=10 6Bytes Address 0x000100 Link Type=ACL
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 01
cmd ogf/ocf opcode plen=7 6 Bytes Address Role=Slave

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

> HCI Event: Connect Complete(0x03) plen 11 - Page 730
04 03 0B 00 00 01 00 04 00 00 00 00 01
evt Connection complete plen=11 status=success 2Bytes(12Bits meaningful) BD_ADDR Link_Type=ACL Encryption_Mode=
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
acl handle/flags dlen = 0x001c the 28 acl data bytes
0x2001 flags = 20 01 >> 12 L2CAP: 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
flags = 0x02 see page 557 handle = 20 01 & Offf (this is used to identify requests / responses)
handle = 0x0001 08 0x08 Code = Echo Request C8 0xc8 Identifier = 2000 14 00 0x0014 Length = 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
acl as above dlen the 28 acl data bytes...
handle/flags 0x001c = 28 L2CAP: 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
18 00 0x0018 Length = 24 01 00 0x0001 Signalling CID 09 0x09 Code Echo Response
09 09 0x09 Code Echo Response C8 0xc8 Identifier = 2000 14 00 0x0014 Length = 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 ctrl+c in the l2ping console stops pinging...

> HCI Event: Disconn Complete(0x05) plen 4 - Page 733
01 05 04 00
cmd disconn complete plen=4 status=disconn has occurred 2Bytes (12Bit Meaningful) connection handle Reason for disconnection=
0x0001 0x0001 connection timeout
08 see table on page 766
```


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

> HCI Event: Command Status(0x0f) plen 4 - Page 745
04 0F 04 00 07
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 01 00
evt code plen=11 status bdaddr link type=ACL encryption mode =disabled
success success connection

> HCI Event: Command Status(0x0f) plen 4 - Page 745
04 0F 04 00 08
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 L2 acl data bytes... L2CAP: cmd data (Page 286)
0x000c = 12 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 00 00
acl handle/flags dlen the 16 acl data bytes L2CAP: cmd data (Page 287)
0x0010 = 16 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
```

< 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... L2CAP cmd data (Page 288)
0x0010 = 16

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
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
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
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
acl handle/flags dlen 14 acl data bytes 7F 7F
0x000E = 14 L2CAP:

0A 00 0x000A Length
40 00 0x0040 Channel ID (not signalling !!!)
25 00 00 00 0A 00 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
^^

* 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 06 0x06 Cmd code (Disconnection Request)
03 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 07 0x07 Cmd Code = Disconnection Response
03 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
09f/ocf 12 bits meaningfull
out of 0x0001

Reason see P 571 = Other End Terminated Connection Error Codes

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

04 0F 04 00 07 06 04

evt code plen=4 command currently pending opcode of command that
no of packets one can still caused event and is 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 occurred out of (0x0001) handle 12 bits
Reason See p 766 Table 6.1
=Connection terminated by local host

> 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

B Demo ping

Bei dieser Demo handelt es sich um folgendes Szenario:

Ein Linux Rechner "pinged" den mobilen AVR Bluetooth Knoten.
Auf der folgenden Seite ist die Ausgabe von `l2ping` des Linux Rechners,
sowie die Ausgabe vom AVR via `minicom` aufgelistet.


```
[root@pc-2711 tools]# ./l2ping 00:d0:b7:03:20:12
Ping: 00:d0:b7:03:20:12 from 00:00:00:00:00:44 (data size 20) ...
20 bytes from 00:d0:b7:03:20:12 id 200 time 138.80ms
20 bytes from 00:d0:b7:03:20:12 id 201 time 180.41ms
20 bytes from 00:d0:b7:03:20:12 id 202 time 180.41ms
20 bytes from 00:d0:b7:03:20:12 id 203 time 200.38ms
20 bytes from 00:d0:b7:03:20:12 id 204 time 180.39ms
20 bytes from 00:d0:b7:03:20:12 id 205 time 190.41ms
20 bytes from 00:d0:b7:03:20:12 id 206 time 170.38ms
20 bytes from 00:d0:b7:03:20:12 id 207 time 180.38ms
8 sent, 8 received, 0% loss
[root@pc-2711 tools]#
```

produces the following output on the serial console (from minicom)

```
-> Please reset ROK (if skt): here we go...
-> Initialising device... finished: 0 out of 10 cmds failed
> Device Info:
    address: 00 d0 b7 03 20 12
    features (see page 243): 47 ea 01 00 00 00 00 00
    sco pkt size: 0x0 [Bytes] max sco pkts: 0x0
    acl pkt size: 0x320 [Bytes] max acl pkts: 0xa
    page timeout: 0x8000
    connection accept timeout: 0x7d00
    number of hci cmds allowed to pass to rok: 0x1

-> Entering inquiry mode...
> EVT_CMD_STATUS
> EVT_INQUIRY_RESULT 03:20:12:04:01:01
> EVT_INQUIRY_COMPLETE
-> Entering listen mode... ready to recv (try xhop -s)

> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=200
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=200
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=201
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=201
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=202
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=202
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=203
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=203
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=204
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=204
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=205
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=205
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=206
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=206
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=207
L2CAP_ECHO_REQ
< L2CAP_ECHO_RSP ident=207
> EVT_DISCONN_COMPLETE (setting l2cap_state to CLOSED)
```

of course this can be stopped, started again and so on...

C Demo xhop

Folgende Seiten zeigen die Ausgabe der Demo "xhop - ohne Kommando Daten"

- Route
(A → B → C)

A sendet ein xhop Paket an B. B leitet das Paket nach C weiter. C ist der Endpunkt und überprüft, ob er Kommandos ausführen soll. Dem ist in diesem Fall nicht so. Schliesslich gibt C das Paket, das er erhalten hat, auf der Konsole aus.

Als erstes sieht man die Ausgabe des Knoten A, danach die Ausgabe von B und C.

starting on a pc the following pkt...

```
[root@pc-2711 xhop]# ./xhop -s
- DEBUG 1267, main -> xhop started in SEND mode

...

print_packet(): showing content of RDSR packet...
  xhop header:
    option type = '1(RDSR_DATA)'
    route len = 0x3
    route pos = 0x1
  route addresses:
    00:00:00:00:00:44
    00:D0:B7:03:20:12
    00:00:00:00:00:55
  xhop data:
    rc length = 8
    prog length = 0
  Answer Data:

- DEBUG 801, send_buffer -> Done ...
- DEBUG 1306, main -> free packet buffer
xhop[9520]: Exit
[root@pc-2711 xhop]#
```

on the avr it looks like:

```
> EVT_CONN_REQUEST
< HCI_ACCEPT_CONN
> EVT_CMD_STATUS
> EVT_CONN_COMPLETE
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=1
  L2CAP_CONN_REQ psm=0xa scid=0x40
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=2
  L2CAP_CONF_REQ dcid=0x40 flags=0x0
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=2
  L2CAP_CONF_RSP scid=0x40 flags=0x0 result=0x0
> L2CAP_DATA cid=0x40 len=29 [psm 10]
> L2CAP_SIGNALLING: ACL handle=0x1 flags=0x2, L2CAP ident=3
  L2CAP_DISCONN_REQ dcid=0x40 scid=0x40
< HCI_DISCONNECT
> EVT_DISCONN_COMPLETE (setting l2cap_state to CLOSED)
-> Analyzing pkt: forwarding
> EVT_CMD_STATUS
> EVT_CONN_COMPLETE
< L2CAP_CONN_REQ
> L2CAP_CONN_RSP handle=0x1 flags=0x2 dcid=0x40 scid=0x40 result=0 status=0
< L2CAP_CONF_REQ
> L2CAP_CONF_RSP
> L2CAP_CONF_REQ handle=0x1 flags=0x2 dcid=0x40 flags=0x0
< L2CAP_CONF_RSP
< DATA
< L2CAP_DISCONN_REQ
> EVT_NUM_COMP_PKTS 5 hci acl datapkts have completed
> L2CAP_DISCONN_RSP
< HCI_DISCONNECT
> EVT_CMD_STATUS
> EVT_DISCONN_COMPLETE
> EVT_CMD_COMPLETE
-> Entering listen mode... (try xhop -s)
```

and is ready again to receive data...

and on the pc that is the last one in this simple chain...

```
[root@pc-3294 /root]# hcidump --hex
HCIDump - HCI packet analyzer ver 11/02/2001.
device: hci0 snap_len: 1024 filter: none
> HCI Event: Connect Request(0x04) plen 10
 12 20 03 B7 D0 00 00 01 00 01
< HCI Command: Accept Connection Request(0x01|0x0009) plen 7
 12 20 03 B7 D0 00 01
> HCI Event: Command Status(0x0f) plen 4
 00 08 09 04
> HCI Event: Connect Complete(0x03) plen 11
 00 01 00 12 20 03 B7 D0 00 01 00
> ACL data: handle 0x0001 flags 0x02 dlen 12
L2CAP(s): Connect req: psm 10 scid 0x0040
< ACL data: handle 0x0001 flags 0x02 dlen 16
L2CAP(s): Connect rsp: dcid 0x0040 scid 0x0040 result 0 status 0
> ACL data: handle 0x0001 flags 0x02 dlen 12
L2CAP(s): Config req: dcid 0x0040 flags 0x0000 clen 0
< ACL data: handle 0x0001 flags 0x02 dlen 14
L2CAP(s): Config rsp: scid 0x0040 flags 0x0000 result 0 clen 0
< ACL data: handle 0x0001 flags 0x02 dlen 12
L2CAP(s): Config req: dcid 0x0040 flags 0x0000 clen 0
> ACL data: handle 0x0001 flags 0x02 dlen 14
L2CAP(s): Config rsp: scid 0x0040 flags 0x0000 result 0 clen 0
> ACL data: handle 0x0001 flags 0x02 dlen 33
L2CAP(d): cid 0x40 len 29 [psm 10]
 01 03 02 00 00 00 00 00 44 00 D0 B7 03 20 12 00 00 00 00 00
 55 08 00 00 00 00 00 00 00 00
> ACL data: handle 0x0001 flags 0x02 dlen 12
L2CAP(s): Disconn req: dcid 0x0040 scid 0x0040
< ACL data: handle 0x0001 flags 0x02 dlen 12
L2CAP(s): Disconn rsp: dcid 0x0040 scid 0x0040
> HCI Event: Number of Completed Packets(0x13) plen 5
 01 01 00 04 00
> HCI Event: Disconn Complete(0x05) plen 4
 00 01 00 13
```

this was in hcidump hex format as it tells more
then just the output from l2test or non-avr-xhop

```
xhop[14943]: Waiting for connection on psm 10 ...
xhop[14944]: Connect from 00:D0:B7:03:20:12 [imtu 672, omtu 672, flush_to 65535]
- DEBUG 672, recv_handler -> allocating buf for receiving packet
data_size=MAX_PACKET_SIZE =640
xhop[14944]: recv_handler() called...
- DEBUG 691, recv_handler -> packet received :
print_packet(): showing content of RDSR packet...
xhop header:
    option type = '1(RDSR_DATA)'
    route len = 0x3
    route pos = 0x2
route addresses:
    00:00:00:00:00:44
    00:D0:B7:03:20:12
    00:00:00:00:00:55
xhop data:
    rc length = 8
    prog length = 0
Answer Data:
- DEBUG 697, recv_handler -> calling analyze()...
- DEBUG 1199, analyze -> analyze() called...
- DEBUG 1200, analyze -> p_len=29 - DEBUG 1218, analyze -> calling exec_cmd...
- DEBUG 154, exec_cmd -> exec_cmd() called
- DEBUG 166, exec_cmd -> Packet without commads recieved -> read answer
```

Literatur

- [1] Lars Wernli und Riccardo Semadeni. Bluetooth unleashed. Semesterarbeit, ETH Zürich, SS 2001.
- [2] Bluetooth Special Interest Group. The official bluetooth specification. <http://www.bluetooth.com/developer/specification/specification.asp>
- [3] Jan Beutel. Geolocation in a picoradio environment. Diplomarbeit ETH Zürich, UC Berkeley, 1999.
- [4] Axis Communication. Axis freier Bluetooth Stack. <http://www.developer.axis.com/software/bluetooth>.
- [5] SourceForge.net. BlueZ - Offizieller Bluetooth Stack für Linux. <http://bluez.sourceforge.net>.
- [6] SourceForge.net BlueZ - Offizieller Bluetooth Stack für Linux. <http://bluez.sourceforge.net/howto/>
- [7] Thomas Moser und Lukas Karrer. The EventCollector Concept. Diplomarbeit, ETH Zürich.
- [8] Oliver Kasten. Forschungsprojekt vom Departement Informatik, ETH Zürich. <http://www.inf.ethz.ch/~kasten/research/smart-its/>
- [9] David B. Johnson, David A. Maltz, Yih-Chun Hu, and Jorjeta G. Jetcheva. The dynamic source routing protocol for mobile ad hoc networks. Internet-draft, Rice University, AON Networks, Rice University, Carnegie Mellon University, <http://www.ietf.org/internet-drafts/draft-ietf-manet-dsr-05.txt>, 2001.

SEMESTERARBEIT

für

Egon Burgener und Peter Fercher

Betreuer: Jan Beutel, ETZ G63

Stellvertreter: Christian Plessl, ETZ G81

Ausgabe: 22. Oktober 2001

Abgabe: 30. Januar 2002

Grenzenlose Piconetze mit Bluetooth**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 jedoch meistens aus einer Basisstation und verschiedenen mobilen Geräten die dann sternförmig miteinander kommunizieren.

Mit Bluetooth lassen sich die unterschiedlichsten Geräte drahtlos verbinden. Dabei können aber immer nur bis zu acht aktive Geräte in einem sogenannten Piconetz miteinander kommunizieren. In einem Raum können mehrere Piconetze in einem Scatternetz koexistieren. Über einen Backbone (z.B. Ethernet) können Bluetooth Geräte in unterschiedlichen Räumen dann miteinander kommunizieren.

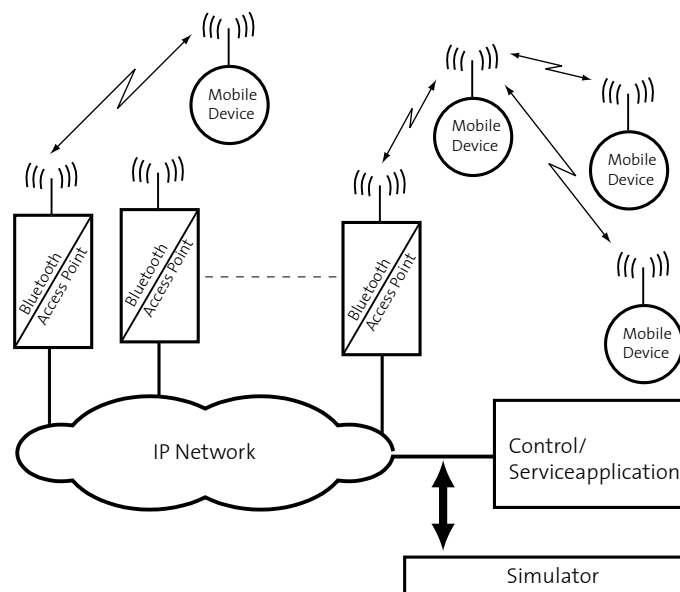


Abbildung 1: Verteilte Bluetooth Access Points und Piconetze

Um viele Bluetooth Geräte zu koordinieren und um ein Netzwerk aus sehr vielen Piconetzen aufzubauen, soll hier aufbauend auf vergangene Semesterarbeiten [23, 24] Steuer und Simulationssoftware entwickelt werden. Ein einfacher Zugriffsmechanismus über Backbone oder mehrere Bluetooth Geräte soll ein vollständig transparentes Netzwerk ermöglichen (in Zusammenarbeit mit der Diplomarbeit "Reconfigurable Bluetooth Ethernet Bridge"). Dabei müssen sowohl Network Discovery und Monitoring wie auch ein einfaches Routing weiterentwickelt werden.

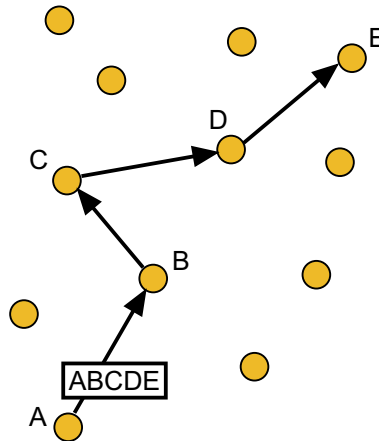


Abbildung 2: Dynamic Source Routing als Multihop Routing

Im Zusammenhang mit dem Smart It's Projekt [1, 25, 14] ist ein miniaturisierter Bluetooth Knoten entstanden, der minimale Protokoll Funktionen ausführen kann. Der BTnode verfügt über ein Bluetooth Modul, einen einfachen Microcontroller [2], In- und Output sowie Powermanagement. Ein einfacher Scheduler bietet die Möglichkeit einfache nebenläufige Routinen auszuführen. Dieser soll nun um eine Multihop Routing Funktion erweitert werden.



Abbildung 3: Bluetooth Node Prototyp mit Bluetooth Module, Microcontroller, und Batterie

Aufgabenstellung

1. Erstellen Sie einen Projektplan und legen Sie Meilensteine fest [27]. Erarbeiten Sie in Absprache mit dem Parallelprojekt "Reconfigurable Bluetooth Ethernet Bridge" und dem Betreuer ein Pflichtenheft.
2. Führen Sie eine Literaturrecherche zu Themen wie Mobile und Ubiquitous Computing, Wireless Networking, Datenlinkprotokolle, und Multihop Netzwerke durch. Ausgangspunkte bilden z.B. die Arbeiten [19, 11, 10, 20, 21, 12, 13]. Suchen Sie auch nach neueren Publikationen.
3. Machen Sie sich mit den am Institut und bei Prof. Mattern (Smart It's) bereits durchgeführten Arbeiten [17, 16, 18, 26, 7, 23, 1] vertraut. Es sollten möglichst viele Synergien aus schon durchgeführten Arbeiten genutzt werden.
4. Arbeiten Sie sich in die Softwareentwicklungsumgebung des Bluetooth Stacks unter Linux ein [15, 4, 5, 9, 8]. Es gibt mehrere Open Source Projekte, führend und im aktuellen Linux Kernel 2.4.x integriert ist BlueZ. Dieser Protokollstack soll in dieser Arbeit zur Verwendung kommen.
5. Implementieren Sie die in der Arbeit von Semadeni und Wernli [23] vorgeschlagene reduzierte variante des Dynamic Source Routing Protokolls für BlueZ.

6. Erweitern Sie die Remote Execution Funktion so das sie zur Überwachung und Messung der Netzwerktopologie benutzt werden kann. Orientieren Sie sich dabei bei den in [3, 22] vorgeschlagenen Mechanismen.

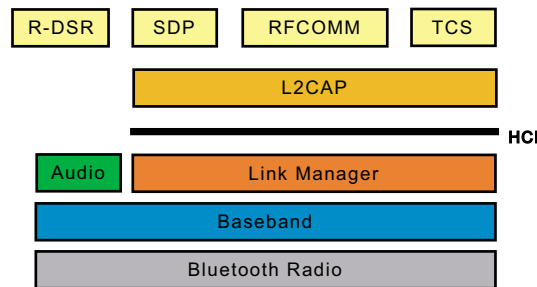


Abbildung 4: Dynamic Source Routing im Bluetooth Protocol integriert.

7. Realisieren Sie eine einfache Oberfläche zur Steuerung und Überwachung eines solchen ad-hoc Netzwerkes.
8. Die Multihop Funktion soll anschliesend auf die BTnode Plattform übertragen werden. Dazu soll sie in ein minimales Betriebssystem wie etwa TinyOS [6] integriert werden.
9. Das in der Semesterarbeit "Multihop Netzwerk Simulator" [24] entwickelte Simulationspaket "Netsim" soll Daten aus dem realen Netzwerk verarbeiten können. Definieren Sie die dazu nötigen Schnittstellen und implementieren Sie diese.
10. Dokumentieren Sie Ihre Arbeit sorgfältig mit einem Vortrag, einer kleinen Demonstration, sowie mit einem Schlussbericht.

Durchführung der Semesterarbeit

Allgemeines

- Der Verlauf des Projektes "Semesterarbeit: Grenzenlose Piconetze mit Bluetooth" soll laufend anhand des Projektplanes und der Meilensteine evaluiert werden. Unvorhergesehene Probleme beim eingeschlagenen Lösungsweg können Änderungen am Projektplan erforderlich machen. Diese sollen dokumentiert werden.
- 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 Ihrem Betreuer.

Abgabe

- Geben Sie vier unterschriebene Exemplare des Berichtes spätestens am 30. Januar 2001 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] The Smart Its Project. <http://www.smart-its.org>.
- [2] Atmel. *Atmel ATmega103L - 8-Bit AVR Microcontroller with 128k in-System programmable Flash*, January 2000.
- [3] Jan Beutel. *Geolocation in a PicoRadio Environment*. Master's thesis, ETH Zürich, Electronics Lab, 1999.
- [4] Jan Beutel and Maksim Krasnyanskiy. *Linux BlueZ Howto*, 2001.
- [5] Axis Communications. *Axis Bluetooth Driver Software*. <http://sourceforge.net/projects/openbt/>.
- [6] David Culler et. al. *TinyOS: An operating system for Networked Sensors*. <http://tinios.millennium.berkeley.edu/>.
- [7] Damon Fenacci, Marcel Grob, Peter Zberg, and Simon Künzli. *Konzept Heim Netzwerk*. Master's thesis, ETH Zürich, TIK, 2000.
- [8] Bluetooth Special Interest Group. *Specification of the Bluetooth System - Core, v.1.1*, February 2001.
- [9] IBM. *BlueDrekar, Bluetooth protocol stack for Linux*. <http://www.alphaWorks.ibm.com/tech/bluedrekar>.
- [10] Thomasz Imielinski and Henry Korth. *Mobile Computing*. Kluwer Academic Publishers, 1996.
- [11] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. *Directed diffusion: A scalable and robust communication paradigm for sensor networks*. Technical Report 00-732, University of Southern California, March 2000.
- [12] A. Iwata, Ching-Chuan Chiang, Guangyu Pei, M. Gerla, and Tsu-Wei Chen. *Scalable routing strategies for ad hoc wireless networks*. *IEEE Journal on Selected Areas in Communications, Special Issue on Ad-Hoc Networks*, 17(8):1369–1379, August 1999.
- [13] Rahul Jain, Anuj Puri, and Raja Sengupta. *Geographical routing using partial information for wireless ad hoc networks*. Technical Report M99/69, UCB/ERL Memo, December 1999.
- [14] Oliver Kasten and Marc Langheinrich. *First experiences with bluetooth in the smart-it's distributed sensor network*. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, September 2001.
- [15] Maksim Krasnyanskiy. *BlueZ, Bluetooth protocol stack for Linux*. <http://bluez.sourceforge.net>.
- [16] H. Mathys. *Bluetooth Stack for the Palmpilot*. Master's thesis, ETH Zürich, TIK, 2001.
- [17] H. Mathys, A. Pellmont, and A. Petralia. *Home Networking - Implementation (Übertitel: DMA - Drop a Message Anywhere)*. Master's thesis, ETH Zürich, TIK, June 2000.
- [18] Thomas Moser and Lukas Karrer. *The EventCollector Concept*. Master's thesis, ETH Zürich, Distributed Systems Group, 2001.
- [19] Danny Patel. *Energy in Ad-Hoc Networking for the PicoRadio*. Master's thesis, University of California at Berkeley, 2000.
- [20] Jan M. Rabaey, Josie Ammer, Julio L. da Silva Jr., Danny Patel, and Shad Roundy. *PicoRadio Supports Ad Hoc Ultra-Low Power Wireless Networking*. *IEEE Computer*, 33(7):42–48, July 2000.
- [21] Volkan Rodoplu and Theresa H. Meng. *Minimum energy mobile wireless networks*. *IEEE Journal on Selected Areas in Communications, Special Issue on Ad-Hoc Networks*, 17(8), August 1999.
- [22] Chris Savarese, Jan Beutel, and Rabaey Jan M. *Locationing in distributed ad-hoc sensor networks*. In *Proceedings of ICASSP*, 2001.
- [23] Riccardo Semadeni and Lars Wernli. *Bluetooth Unleashed, Wireless Netzwerke ohne Grenzen*. Master's thesis, ETH Zürich, TIK, July 2001.
- [24] Reto Strahl. *Build your own world*. Master's thesis, ETH Zürich, TIK, July 2001.

- [25] Jan Beutel und Oliver Kasten. A minimal bluetooth-based computing and communication platform. Technical report, ETH Zürich, May 2001.
- [26] Sascha Wulff and Marco Wirz. Bluetooth Networking - Protocol and Application Development. Master's thesis, ETH Zürich, TIK, February 2001.
- [27] Eckart Zitzler. Studien- und Diplomarbeiten, Merkblatt für Studenten und Betreuer. ETH Zürich, TIK, March 1998.

Date	Section	Changes
Sept. 25, 2001		Initial Version
Oct. 2, 2001		New Stylesheet
Oct. 22, 2001		Specify Workpackage

Tabelle 1: Revision History