UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

AN INVESTIGATION INTO THE VIABILITY OF UWB AS LOWER-LAYER FOR BLUETOOTH

by

Etienne van der Linde

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Computer Engineering)

in the

Faculty of Engineering, the Built Environment and Information Technology

UNIVERSITY OF PRETORIA

August 2009

ACKNOWLEDGEMENTS

A special word of thanks to my supervisor, Prof. Gerhard Hancke of the Department of Electrical, Electronic and Computer Engineering at the University of Pretoria for his support and guidance throughout this research.

The author would like to thank Iñaky Pérez-González and Ed Stock of Intel, Anderson Lizardo of the Nokia Research Institute and the other developers of the Linux UWB driver for the debug support provided during this project implementation.

I would also like to thank my family and loved ones, who supported and motivated me throughout the research process.

AN INVESTIGATION INTO THE VIABILITY OF UWB AS LOWER-LAYER FOR BLUETOOTH

by

Etienne van der Linde

Supervisor:	Prof. G.P. Hancke
Department:	Electrical, Electronic and Computer Engineering
Degree:	Master of Engineering (Computer Engineering)

KEYWORDS:

Ultra Wideband, Bluetooth ECMA-368, UWB, protocol, BToUWB, WPAN, simulation.

ABSTRACT

This report presents an investigation into some merging options between an upper-layer Bluetooth (BT) protocol stack with a lower-layer ECMA-368/9 Ultra Wideband (UWB) radio connection. A Bluetooth over Ultra Wideband (BToUWB) system is implemented by channelling an existing compliant Bluetooth connection's data over an Ultra Wideband Medium Access Control (MAC) and Physical (PHY) layer radio channel. The aim of this project is to provide a description of the methodology used to create a BToUWB link and evaluate some advantages pertaining to the merger between the two Wireless Personal Area Network (WPAN) technologies.

Prior to channelling data over a UWB connection, a compliant Bluetooth and UWB connection were configured between two Linux enabled computers by use of Bluetooth and UWB enabled Universal Serial Bus (USB) dongles. BlueZ, the official Bluetooth stack for Linux, were used to implement a modified Bluetooth stack. By modifying the open source BlueZ files, the Host Controller Interface (HCI) commands sent to the HCI sublayer by upper layer Logical Link Control and Adaptation Protocol (L2CAP) and Synchronous Connection-Oriented (SCO) implementations were hijacked and routed to a UWB "router and convergence" implementation for transmission over the UWB subsytem. Similarly lower

level HCI events were spoofed to the L2CAP and SCO layers by the UWB convergence implementation upon receiving packets from the UWB subsystem.

The commercial availability of UWB hardware through Wireless USB dongles enabled the realization of a compliant UWB link between the systems, requiring special driver modifications and Intel provided firmware to establish a WiMedia Logical Link Control Protocol (WLP) network. A specially developed test program generates L2CAP, Radio Frequency Communication (RFCOMM) and SCO Bluetooth data for testing the BToUWB link. The various Bluetooth data packets are routed from the Bluetooth stack to a developed kernel space routing module, which encapsulated the packets and route them via the WLP interface over the wireless high-speed UWB network to the remote system. On the remote side, the packets propagate its way back up through the UWB hardware and software module, and to the router module via call-back functions in the WLP interface. The router module strips the headers and injects the packets back into the Bluetooth L2CAP, RFCOMM or SCO layer for further Bluetooth processing. A test program running on the remote system, receives the test data and loops it back for asynchronous analyses, or stores it for later comparison in synchronous analyses.

The results obtained from the system analyses shows how a Bluetooth system can benefit from implementing UWB as lower layer wireless interface over a short range by either improved asynchronous bandwidth, or synchronous reliability. The results also show some limitations of the pilot UWB hardware and firmware available over longer distances. In general, the successful transmission of Bluetooth data over the BToUWB implemented system proves the HCI layer to be a viable mergence point between the two protocols.

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

‘N ONDERSOEK NA DIE GANGBAARHEID VAN ULTRA WYEBAND AS LAER VLAK KOMMUNIKASIE MEDIUM VIR DIE BLUETOOTH-PROTOKOL

deur

Etienne van der Linde

Studieleier:	Prof. G.P. Hancke
Departement:	Elektriese, Elektroniese en Rekenaar-Ingenieurswese
Graad:	Meester van Ingenieurswese (Rekenaar-Ingenieurswese)

SLEUTELWOORDE:

Ultra-Wyeband, Bluetooth, ECMA-368, UWB, protokol, BToUWB, KPAN, simulasie.

OPSOMMING

Hierdie verslag bied 'n studie van die versoeningspunte tussen 'n hoër-vlak Bluetooth-protokol en laer-vlak ECMA-368/9 Ultra-Wyeband (UWB) -radiokonneksie. 'n Bluetooth-oor-Ultra-Wyeband (BToUWB) stelsel is geïmplementeer deur bestaande Bluetoothdata oor 'n Ultra-Wyeband Medium Toegangsbeheer (MT) en fisiese radiokanaalvlak te kanaliseer. Die doelwit van die projek is om 'n beskrywing te bied van die metodologie wat gevolg is om 'n BToUWB-skakel te implementeer en voordele wat die versoening tussen die twee Koordlose Persoonlike Area Netwerk (KPAN) -tegnologieë inhou te evalueer.

Voordat data oor die UWB-konneksie gekanaliseer kan word, moet Bluetooth- en UWB-konneksies eers gekonfigureer word tussen twee Linux bevoegde rekenaars deur die gebruik van Bluetooth- en UWB-Universele Seriale Bus (USB) -toestelle. Bluez, die offisiële Bluetooth-stapel vir Linux, was gebruik om 'n gemodifiseerde Bluetooth-protokol vir die sisteem te implementeer. Die Gasheer Kontroleerder-Koppelvlak (GKK) opdragte wat deur hoër-vlak Logiese Skakelbeheer-en-Aanpassings Protokol (LSAP) en Sinkrone Konneksie-Georiënteerde (SKG) -implementasies gestuur is na die GKK sub-vlak is onderskep en gekanaliseer na 'n UWB “stuur-en-aansluiting” -implementasie vir versending oor die UWB sub-sisteem. Soortgelyk hieraan is laer-vlak GKK gebeure vervals en aangestuur na die LSAP en SKG vlakke deur die “stuur-en-aansluiting” -implementasie wanneer datapakkies vanaf die UWB sub-sisteem ontvang is.

Die kommersiële beskikbaarheid van UWB-hardeware deur draadlose USB-toestelle het die realisering van 'n voldoende UWB-skakel tussen die sisteme moontlik gemaak. Hierdie toestelle het spesiale drywer modifikasies en Intel-voorsiende sagteware vereis om ‘n WiMedia-Logiese-Skakelbeheer-Protokol (WLSP) -netwerk te bewerkstellig. 'n Toetsprogram is spesiaal ontwikkel om LSAP, Radio-Frekwensie-Kommunikasie (RFKOMM) en SKG Bluetooth-data te genereer om die BToUWB-skakel te toets. Die verskeie Bluetooth-datapakkies word dan gestuur vanaf die Bluetooth-stapel na 'n ontwikkelde Linux “Kernel”-basis module, waar die pakkies herverpak en via die WLSP-koppelvlak oor die draadlose hoë-spoed UWB netwerk gestuur word na die afgeleë sisteem. Aan die afgeleë kant word die pakkies terug op gepropageer deur die sisteem via die UWB-hardeware- en sagteware-modules, tot en met die “stuur-en-aansluiting” -module via terugroep-funksies in die WLSP koppelvlak. Die “stuur-en-aansluiting” -module stroop die datapakkies se kopskrifte, en spuit dan die pakkies terug binne in die Bluetooth LSAP, RFKOMM of SKG vlakke vir verdere Bluetooth prosessering. 'n Toetsprogram wat op die afgeleë sisteem loop, ontvang die toetsdata en versend dit terug na die oorspronklike sisteem vir asinkrone analisering, of stoor dit vir latere vergelyking in sinkrone analisering.

Die resultate verkry vanaf die data analisering toon hoe 'n Bluetooth sisteem voordeel kan trek uit die implementering van UWB as laer-vlak draadlose koppelvlak oor 'n kort afstand deur of verbeterde asinkrone bandwydte, of sinkrone betroubaarheid. Die uitslae toon ook sekere beperkings van die eerste beskikbare UWB-hardeware en -sagteware oor langer afstande. Oor die algemeen het die suksesvolle versending van Bluetooth-data oor die BToUWB geïmplementeerde sisteem bewys dat die GKK vlak 'n lewensvatbaar aansluitingspunt tussen die twee protokolle is.

LIST OF ABBREVIATIONS

ACK	Acknowledgement
ACL	Asynchronous Connectionless
AES	Advanced Encryption Standard
AM_ADDR	Active Member Address
AMP	Alternative MAC/PHY
API	Application Programming Interface
AR_ADDR	Access Request Address
ARQ	Automatic Repeat-reQuest
ARQN	Automatic Repeat-reQuest Number
ASCII	American Standard Code for Information Interchange
BB	Baseband
BD_ADDR	Bluetooth Device Address
BER	Bit Error Rate
BNEP	Bluetooth Network Encapsulation Protocol
BP	Beacon Period
BPST	Beacon Period Start Time
BT	Bluetooth
BToUWB	Bluetooth over Ultra-Wideband
CAC	Channel Access Code
CAPI	Common Application Program Interface
CCA	Clear Channel Assessment
CD	Carrier Detect
CDMA	Collision Detect Multiple Access
CID	Channel Identification number
CMTP	CAPI Message Transport Protocol
CRC	Cyclic Redundancy Code
CSMA	Carrier Sense Multiple Access
CTP	Cordless Telephony Profile
CTS	Clear to Send
CVSD	Continuously Variable Slope Delta modulation
DAC	Device Access Code
dBm	Decibel (referenced to milliwatts)
DCE	Data Communications Equipment
DH	Data High rate
DISC	Disconnect
DLC	Data Link Connection
DLCI	Data Link Connection Identifier

DM	Data Medium rate
DRP	Distributed Reservation Protocol
DRP	Distributed Reservation Protocol
DSR	Data Set Ready
DTI	Data Transfer In
DTO	Data Transfer Out
DTR	Data Terminal Ready
DUNP	Dial-up Networking Profile
DV	Data Voice combined
ECMA	European Computer Manufacturers' Association
EDA	External Data Access
EHCI	Enhanced Host Controller Interface
eSCO	extended Synchronous Connection-Oriented
ETSI	European Telecommunications Standards Institute
EUI	Extended Unique Identifier
FCC	Federal Communications Commission
FCS	Frame Check Sequence
FEC	Forward Error Correction
FFI	Fixed Frequency Interleaving
FHS	Frequency Hop Synchronisation
FIFO	First In First Out
FP	Fax Profile
FTP	File Transfer Profile
GAP	Generic Access Profile
GCC	GNU C Compiler
GFSK	Gaussian Frequency Shift Keying
GHz	Gigahertz
GND	Ground
GNOME	GNU Network Object Model Environment
GPL	General Public License
GTK	Group Temporal Key
GUI	Graphical User Interface
HC	Host Controller
HCI	Host Controller Interface
HCS	Header Check Sequence
HID	Human Interface Device
HIDP	Human Interaction Design Protocol
HS	Headset Profile
HV	High quality Voice

HWA	Host Wire Adapter
IAC	Inquiry Access Code
ID	Identification number
IE	Information Element
IEEE	Institute of Electrical & Electronics Engineers
IH	Information Header
IP	Internet Protocol
IP	Intercom Profile
ISM	Industrial, Scientific and Medical (radio spectrum)
kbps	kilobytes per second
KDE	K Desktop Environment
kHz	kilohertz
L2CAP	Logical Link Control and Adaptation Protocol
LAN	Local Area Network
LAP	Lower Address Part
LAP	LAN Access Profile
LC	Link Controller
LLC	Logical Link Control
LLID	Logical Link Identification
LM	Link Manager
LMP	Link Manager Protocol
LQE	Link Quality Estimate
LSB	Least Significant Bit/Byte
MAC	Medium Access Control
MAS	Medium Access Slot
MB	Multiband
MBOA	Multiband-OFDM Alliance
Mbps	megabits per second
MHz	Megahertz
MIC	Message Integrity Code
MSB	Most Significant Bit/Byte
MSDU	MAC Service Data Unit
MTU	Maximum Transmission Unit
MUX	Multiplexer
NAP	Nonsignificant Address Part
OBEX	Object Exchange
OBEXP	OBEX Profile
OCF	Opcode Command Field
OFDM	Orthogonal Frequency Division Multiplexing

OGF	Opcode Group Field
OPP	Object Push Profile
OUI	Organisationally Unique Identifier
PCA	Prioritised Contention Access
PCB	Printed Circuit Board
PCM	Pulse-Code Modulation
PCMCIA	Personal Computer Memory Card International Association
PDA	Personal Digital Assistant
PDU	Protocol Data Unit
PHY	Physical interface
PIM	Personal Information Management
PIN	Personal Identification Number
PLCP	Physical Layer Conversion Protocol
PLCP	Physical Layer Convergence Protocol
PLME	Physical Layer Management Entity
PM_ADDR	Parked Member Address
PMD	Physical Medium Dependent
PN	Parameter Negotiation
PNP	Plug and Play
PPP	Point-to-Point Protocol
PSDU	Physical layer Service Data Unit
PSM	Protocol Service Multiplexing
QoS	Quality of Service
RC	Radio Controller
RD	Receive Data
RF	Radio Frequency
RFCOMM	Radio Frequency Communication
RI	Ring Indicator
RLS	Remote Line Status
RPN	Remote Port Negotiation
RS232	Recommended Standard 232
RSSI	Receiver Signal Strength Indicator
RTS	Request to Send
RX	Receiver
SABM	Start Asynchronous Balanced Mode
SAR	Segmentation and Reassembly
SCO	Synchronous Connection-Oriented
SDAP	Service Discovery Application Profile
SDP	Service Discovery Protocol

SIG	Special Interest Group
SP	Synchronisation Profile
SPP	Serial Port Profile
TCP	Transmission Control Protocol
TCS-BIN	Telephony Control Signalling for Binary
TD	Transmit Data
TDD	Time Division Duplex
TDMA	Time Division Multiple Access
TFI	Time Frequency Interleaving
TPK	Temporal Key
TTY	Terminal Type
TX	Transmitter
UA	Asynchronous User information channel
UA	Unnumbered Acknowledgement
UAP	Upper Address Part
UART	Universal Asynchronous Receiver-Transmitter
uCLinux	Micro Linux
UDP	Unreliable Datagram Protocol
UI	Isochronous User information channel
UIH	Unnumbered Information with Header
URB	USB Request Block
US	United States
us	micro seconds
US	Synchronous User information channel
USB	Universal Serial Bus
UTF	Universal Transaction Format
UUID	Universal Unique Identifier
UWB	Ultra-Wideband
VHCI	Virtual HCI
WAE	WAP Application Environment
WAP	Wireless Application Protocol
WLAN	Wireless Local Area Network
WLP	WiMedia LLC Protocol
WPAN	Wireless Personal Area Networks
WSS	WiNet Service Set
WUSB	Wireless Universal Serial Bus

TABLE OF CONTENTS

CHAPTER 1 : INTRODUCTION 1
1.1 SCOPE 1
1.2 MOTIVATION 2
1.3 OBJECTIVES 3
1.4 RESEARCH METHODOLOGY 3
1.5 CONTRIBUTION 4
1.6 DOCUMENT STRUCTURE 4
CHAPTER 2 : OVERVIEW OF CURRENT LITERATURE 5
2.1 BLUETOOTH 5
2.1.1 Bluetooth Protocol Stack 5
2.1.1.1 Radio 6
2.1.1.2 Baseband 6
2.1.1.2.1 Physical Channel 6
2.1.1.2.2 Device Addressing 7
2.1.1.2.3 Physical Links 8
2.1.1.2.4 Logical Channels 8
2.1.1.2.5 Packets 9
2.1.1.2.6 Channel Control 10
2.1.1.2.7 Error Correction 12
2.1.1.2.8 Flow Control 12
2.1.1.2.9 Synchronising 12
2.1.1.2.10 Security 13
2.1.1.3 Link Manager Protocol (LMP) 13
2.1.1.4 Host Controller Interface (HCI) 13
2.1.1.5 Logical Link Control and Adaptation Protocol (L2CAP) 15
2.1.1.5.1 L2CAP Channelling 16
2.1.1.5.2 L2CAP Segmentation and Reassembly 17
2.1.1.6 Synchronous Audio Data 17
2.1.1.6.1 Synchronous Connection Oriented (SCO) Logical Link 17
2.1.1.6.2 Extended Synchronous Connection Oriented (eSCO) Logical Link 17
2.1.1.7 Service Discover Protocol (SDP) 18
2.1.1.7.1 SDP Overview 18
2.1.1.7.2 SDP Services 19
2.1.1.7.3 Service Discovery 19
2.1.1.8 RFCOMM 20
2.1.1.8.1 Overview 20
2.1.1.8.2 TS 07.10 adaptations 21
2.1.1.8.3 Flow Control Methods 23
2.1.1.9 Application Layer 23
2.1.2 Bluetooth Profiles 24
2.2 ULTRA WIDEBAND 26
2.2.1 ECMA UWB Standards 27
2.2.1.1 ECMA-368 High Rate UWB PHY and MAC Standards 27
2.2.1.1.1 UWB PHY Specification 28
2.2.1.1.1.1 PMD Sublayer 29
2.2.1.1.1.2 PLCP Sublayer 29
2.2.1.1.1.3 PLME Sublayer 29

2.2.1.1.1.4 Signal Description 29
2.2.1.1.1.5 Frame Structure 30
2.2.1.1.2 UWB MAC Specification 30
2.2.1.1.2.1 Medium Access Schemes 31
2.2.1.1.2.2 MAC Addressing 32
2.2.1.1.2.3 Device Discovery and Channel Selection 32
2.2.1.1.2.4 Logical groups 32
2.2.1.1.2.5 Timing and Asynchronous Data Transfer 33
2.2.1.1.2.6 Reliability 33
2.2.1.1.2.7 Security 33
2.2.1.1.2.8 Service Discovery 33
2.2.1.1.2.9 Power Management 34
2.2.1.1.2.10 MUX sublayer 34
2.2.1.2 ECMA-369 MAC-PHY Interface Standard for ECMA-368 34
2.3 BLUETOOTH MERGENCE WITH UWB 36
CHAPTER 3 : DESIGN AND IMPLEMENTATION 38
3.1 SYSTEM OVERVIEW 38
3.1.1 Bluetooth over UWB Simulation 38
3.1.2 Bluetooth over UWB System Implementation 40
3.2 COMPONENT DESCRIPTION 43
3.2.1 BlueZ Protocol Stack 43
3.2.1.1 BlueZ setup 44
3.2.1.2 BlueZ tools and utilities 45
3.2.2 Bluetooth programming 46
3.2.2.1 Bluetooth data generator and analysis program 47
3.2.3 Linux UWB Stack 48
3.2.3.1 Requirements 48
3.2.3.2 Device Discovery 49
3.2.3.3 Data Transmission 49
3.2.3.4 UWB Driver Implementation 50
3.2.4 UWB Simulation subsystem 50
3.2.5 UWB Routing 53
3.2.5.1 Bluetooth Connection 56
3.2.5.2 UWB WLP Connection 56
3.2.5.3 UWB Link Control User Application 57
3.2.6 WLP Convergence 57
3.3 ANALYSIS 59
3.3.1 Bluetooth Asynchronous Data Analysis 59
3.3.1.1 L2CAP ACL Test Data Acquisition 60
3.3.1.2 RFCOMM Test Data Acquisition 61
3.3.2 Bluetooth Isochronous Data Analysis 62
3.3.2.1 SCO Test Data Acquisition 62
CHAPTER 4 : RESULTS 65
4.1 BLUETOOTH OVER UWB SIMULATION 65
4.2 BLUETOOTH OVER UWB SYTEM IMPLEMENTATION 66
4.2.1 Bluetooth Asynchronous Data Analysis 66
4.2.1.1 L2CAP ACL Data : Bandwidth vs Packet Size 66
4.2.1.2 L2CAP ACL Data : Bandwidth vs Transmission Distance 69
4.2.1.3 RFCOMM Data: Bandwidth vs Packet Size 71
4.2.1.4 RFCOMM Data: Bandwidth vs Transmission Distance 73
4.2.1.5 Data Discontinuity 75

4.2.1.6 Bandwidth Decay 76
4.2.1.7 Theoretical Application Throughput 76
4.2.2 Bluetooth Isochronous Data Analysis 78
CHAPTER 5 : CONCLUSION 80
5.1 SUMMARY 80
5.2 BREAKDOWN 80
5.2.1 Bluetooth over UWB Simulation 80
5.2.2 Bluetooth over UWB System Implementation 81
5.2.2.1 L2CAP Asynchronous Data 81
5.2.2.2 RFCOMM Asynchronous Data 81
5.2.2.3 SCO Isochronous Data 82
5.2.3 Bluetooth mergence with UWB 82
5.3 FUTURE CONTRIBUTIONS 83
REFERENCES 84
ADDENDUMS 1
A BLUEZ AND GENERIC UTILITIES 1
A.1 HCICONFIG 1
A.1.1 Description 1
A.1.2 Usage 1
A.1.3 Examples 2
A.2 HCIATTACH 2
A.2.1 Description 2
A.2.2 Usage 2
A.2.3 Examples 2
A.3 HCITOOL 2
A.3.1 Description 2
A.3.2 Usage 3
A.3.2.1 HCI Monitoring 3
A.3.2.2 HCI Writing 3
A.3.3 Examples 3
A.4 HCIDUMP 4
A.4.1 Description 4
A.4.2 Usage 4
A.4.3 Example 5
A.5 HCIEMU 5
A.5.1 Description 5
A.5.2 Usage 5
A.5.3 Example 5
A.6 L2PING 5
A.6.1 Description 5
A.6.2 Usage 5
A.6.3 Example 6
A.7 L2TEST 6
A.7.1 Description 6
A.7.2 Usage 6
A.7.3 Examples 7
A.8 SCOTEST 7
A.8.1 Description 7
A.8.2 Usage 7
A.9 RFCOMM 7
A.9.1 Description 7

A.9.2 Usage 7
A.9.3 Example 8
B BLUETOOTH PROGRAMMING 9
B.1 Common Bluetooth variables and data structures 9
B.2 Common Bluetooth functions 11
B.3 Sample Bluetooth user program 14
B.4 Bluetooth common socket interface module 19
B.4.1 Btapi Header File 19
B.4.2 Btapi Source File 19
B.5 Bluetooth data generator and test program source 28
C UWB UTILITIES 39
C.1 Connection Utilities 39
C.2 WLP Utilities 40
D BToUWB SYSTEM SOURCE FILES 43
D.1 UWB Router Kernel Module 43
D.1.1 UWB Router Kernel Module Header File 43
D.1.2 UWB Router Kernel Module Source File 44
D.2 UWB WLP Convergence Module 53
D.2.1 UWB WLP Convergence Module Header File 53
D.2.2 UWB WLP Convergence Module Source File 53
D.3 UWB Link Control User Application 64
D.3.1 UWB Link Control User Application Source Files 64
D.4 Bluetooth Stack Modifications 68
D.4.1 BlueZ l2cap.c modifications 68
D.4.2 BlueZ sco.c modifications 72

CHAPTER 1: INTRODUCTION

1.1 SCOPE

Bluetooth technology provides convenience and flexibility by replacing the cumbersome cords that connects electronic devices together with an invisible, low power short-range wireless connection and allows the equipment to move throughout the nearby proximity [1]. It is an open standard that provides an ad-hoc approach for enabling various devices to communicate with one another in a nominal 10 meter range. Bluetooth operates in the 2.4 GHz ISM Band (2400-2483.5 MHz) and uses a frequency hopping TDD scheme for each radio channel [2]. The Bluetooth physical-layer has been designed to support operation in very high interference levels, but it is also the bottleneck in the IEEE 802.11b specification, limiting the data throughput of Bluetooth to 721 kbps [3], [4].

Ultra Wideband (UWB) is a wireless technology promising higher data rates than Bluetooth whilst requiring less power. UWB is a “newer” technology in the sense that the U.S. Federal Communications Commission (FCC) has only approved it as recently as April 2002 due to substantial changes in the U.S. radio policy. Interference concerns from incumbent users such as the U.S. Department of Defence kept UWB in research mode for more than 3 year. So far it is only approved for indoor and short-range outdoor communications in the 3 – 10 GHz band with the lowest emission levels ever set for a system [5], [6], [7]. Contrary to the conventional narrow band transmission of radio frequency (RF) signals for easy separation of communication services, UWB uses transmissions that have a wider spectrum spread than any other RF communications system [6]. Instead of mixing code symbols with data to spread the bandwidth of the signal or modulating pulses in terms of phase and amplitude, receiving UWB signals is about measuring the time difference between the arrival of pulses.

The prospective low power and high bandwidth offered over short range has thus far seen Ultra Wideband being implemented as the radio-technology in the new Wireless USB, as well as prototype hardware applications for short-range wireless video streaming. Originally it was anticipated that Ultra Wideband would replace Bluetooth as the dominant short-range wireless technology in the market, but Bluetooth’s commercial stronghold ensured its survival. In 2007 the Bluetooth SIG announced its intensions to incorporate Ultra Wideband as ad-on high-speed radio technology in the Bluetooth 4 release due 2010. In the meanwhile, the Bluetooth 3.0 specification released on 21 April 2009 incorporated an Alternative MAC/PHY (AMP) feature, which enables the use of an alternative high-speed radio for transporting Bluetooth profile data, whilst the Bluetooth radio is still used for device discovery and connection setup. Against expectation, the Bluetooth 3.0 AMP specification currently only supports an 802.11 (Wi-Fi) MAC PHY, providing Bluetooth with higher speed of 24 Mbps over longer distances. The AMP feature however still paves the way for incorporation of the low-power high-bandwidth UWB radio technology. In March 2009 the WiMedia Alliance announced its intention to disband and transfer all current and future specifications to the Bluetooth SIG, Wireless UWB Promoter Group and USB Implementers Forum. This is due to the high overlap between Bluetooth SIG and WiMedia Alliance members, together with the need for UWB to align itself with technologies in momentum, instead of trying to create its own during a recession period. This move will however cease the existence of a singular point of specification for UWB, resulting in the disadvantage of two different UWB versions, whilst allowing the advantage of optimising the protocol for a specific Bluetooth or USB technology.

1.2 MOTIVATION

Combining the upper layers of the well-established Bluetooth protocol with the physical-layer of Ultra Wideband may provide a number of advantages to the Bluetooth protocol stack:

- Increased transfer speeds for asynchronous Bluetooth data transmissions.
- Increased robustness for Bluetooth isochronous transfers.
- Less power consumption required for Bluetooth enabled devices.
- Increased wireless transfer distances between remote devices.

Advantages provided to Ultra Wideband by the merger includes:

- Access to proven Bluetooth wireless techniques, including device discovery and security management.
- Exposure to an established wireless device market adapted to the Bluetooth interface.
- Facilitation of numerous wireless applications needing little or no adaptation for Ultra Wideband functioning.

1.3 OBJECTIVES

The viability of a merger between Bluetooth and Ultra Wideband was investigated by developing and implementing a Bluetooth over Ultra Wideband data channelling system with an emphasis on the design of such a merged system and the viability of the proposed merger points. The system was then applied to transfer Bluetooth data over Ultra Wideband to prove its functioning and unveil advantages it may provide, especially with regards to speed and distance.

1.4 RESEARCH METHODOLOGY

The Bluetooth and Ultra Wideband technologies were studied with respect to their functionality, accessibility and performance. Research was undertaken into the following concepts to investigate the viability of the proposed merger and the different available implementations of each protocol to be applied in the intended developed merged system;

- Bluetooth protocol functionality and interaction between the different Bluetooth protocol layers. This included an investigation into the dataflow between two remote Bluetooth connected applications through the Bluetooth protocol stack.
- Ultra Wideband PHY, MAC and radio-controller functionality and interaction.
- Selection between the accessible variations of the two protocols best suited for the proposed development of a merged system, including a subset of Bluetooth functionality that is feasible for implementation over a UWB physical connection.
- Feasible and affordable acquisition of the new ‘protected’ Ultra Wideband technology and successful implementation of a reliably modifiable Ultra Wideband system.
- Selection of feasible points of fusion between the two protocol stacks.
- Development and analyses of the glue-layer functionality required for successful fusion between the two protocols with the selected subset of Bluetooth functionality.
- Analyses of enhancements produced by the developed Bluetooth over Ultra Wideband system coinciding the two protocols.

1.5 CONTRIBUTION

Although Ultra Wideband and Bluetooth have been compared with each other on a number of levels, the literature study revealed that little integration efforts between UWB and Bluetooth have been published yet, despite the Bluetooth Special Interest Group's (SIG) announcement to incorporate UWB as its high-speed wireless alternative radio technology.

The research performed focuses on fusing these two wireless technologies on a feasible level, and explore the potential that such a merger may disclose. Viably integrating these protocols may combine the advantages of Bluetooth as a well established widely implemented technology with that offered by UWB such as speed and distance. Ultimately it may provide upper layer based Bluetooth devices with the bridging functionality to exploit the advantages offered by Ultra Wideband.

1.6 DOCUMENT STRUCTURE

The theoretical research performed is summarised in Chapter 2 with a rundown of both the Bluetooth and Ultra Wideband protocol stack functionality and interaction. The chapter is concluded with findings pertaining the viability of a fusion between these two protocols.

Chapter 3 elucidates the practical research performed by providing a detailed description of the developed system merging the two protocols, including justification for certain merging selections made and functional explanation of the various system components. Finally the different analysis performed on the developed system is described in section 3.3.

The results obtained from the abovementioned analysis are portrayed in Chapter 4, with the assumptions concluded from it provided in Chapter 5.

Finally the addendum provides detailed information to reproduce the developed system, including applied utilities and developed source code.

CHAPTER 2: OVERVIEW OF CURRENT LITERATURE

Prior to attempting any kind of solution, an in-depth knowledge of the two protocol systems, including its functioning and dataflow, needed to be acquired. A snapshot overview of the Bluetooth and Ultra Wideband protocol stacks at the time of writing this document is provided in this chapter. Section 2.3 covers some initial investigation into the mergence between these two protocol stacks.

2.1 BLUETOOTH

Bluetooth is a wireless protocol for establishing short range frequency-hopping radio links between enabled devices, with robustness, low complexity, low power and low cost as key features. Its aim is to replace traditional cable connections between devices with a universal radio link.

2.1.1 Bluetooth Protocol Stack

Bluetooth is a layered protocol stack, with each layer aiming to provide transparent functionality to its upper layer. Figure 1 shows an overview of the layered stack. The following section will give a brief description of the relevant layers' functionality.

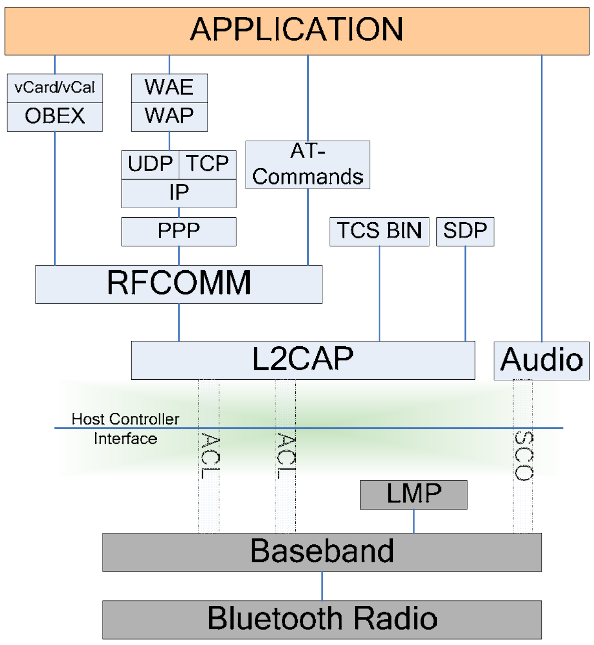

Figure 1: Bluetooth protocol layers [16]

2.1.1.1 Radio

Bluetooth radio modules operate in the unlicensed 2.4 GHz ISM band, and avoid interference from other signals by using a spectrum spreading technique of frequency hopping. The protocol divides the frequency band between 2.402 GHz and 2.480 GHz into 79 equal sub-bands of 1 MHz each, with a guard band used at the lower and upper edge. Frequency tolerance is set at ±75 kHz of the initial centre frequency. A pseudo-random hopping sequence is used to hop through the 79 available channels at a rate of 1.6 kHz.

The Bluetooth radio module uses GFSK (Gaussian Frequency Shift Keying) as modulation technique for radio transmission. This involves representing a binary one by a positive frequency deviation and a zero by a negative frequency deviation.

Bluetooth transmitters are classified into three power classes, shown in Table 1. Devices that can vary their transmitted power are allowed to optimise it by using LMP commands generated by a receiving device, measuring its received signal strength by making use of the Receiver Signal Strength Indicator (RSSI).

Class	Range	Max output power
Class 1	100 m	20 dBm
Class 2	10 m	4 dBm
Class 3	100 mm	0 dBm

Table 1: Transmitter Power Classes

On the receiver side the sensitivity level must be above -70 dBm, which means a bit error rate (BER) of 0.1% is met.

2.1.1.2 Baseband

The baseband physical layer implements the Bluetooth Link Controller (LC) which manages physical channels and links whilst providing services like error correction, data whitening, hop selection and security. The baseband also manages asynchronous and synchronous links, handles packets, and performs inquiry and paging to discover and access Bluetooth devices in the area.

2.1.1.2.1 Physical Channel

Two or more Bluetooth devices linked on the same channel form a piconet, with one device taking the role of master, and the rest as slaves. A piconet can only have one master, but multiple slaves. A device can however be a slave in one piconet whilst being a master or a slave in another, forming what we call a scatternet better illustrated in Figure 2.

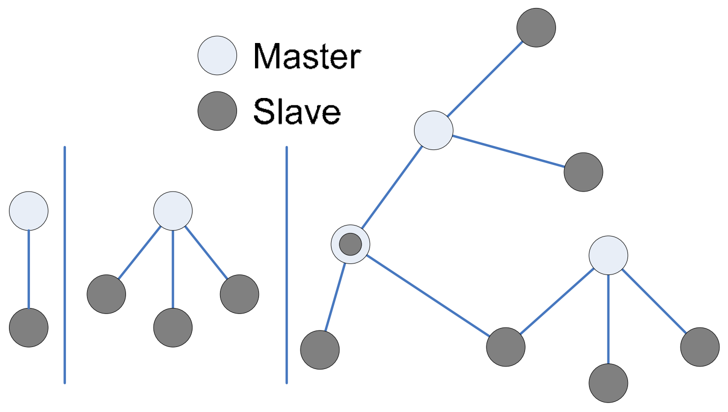

Figure 2: Bluetooth piconets and scatternets [17]

As described in section 1.1.1.1, the Bluetooth radio layer achieves robustness by hopping to another one of the 79 available transmission channels once every 625 micro seconds. Two linking Bluetooth devices transmit on the same channel and will follow the same unique hopping sequence based on a pseudo-random sequence derived from the master's Bluetooth Device Address (BD_ADDR). The phase of the hopping sequence is determined by the Bluetooth clock of the master.

The baseband transceiver uses a Time Division Duplex (TDD) scheme to alternate transmitting and receiving signals over the radio channel, with each timeslot corresponding to a change in frequency (a frequency hop). The physical channel is thus divided into timeslots of 625 us in length, numbered according to the Bluetooth clock of the master. A master shall only start its transmission in even-numbered time slots, and a slave in odd-numbered time slots. A packet start is aligned with a slot start.

When multiple piconets exist in the same area, a single Bluetooth device can participate in two or more overlaying piconets by applying time multiplexing. It should also use the appropriate master address and clock offset to obtain the correct phase, channel hopping sequence and channel access code (CAC). A device can participate as slave in several piconets, but only as master in a single piconet. A group of piconets connected in this way is called a scatternet, also illustrated in Figure 2.

2.1.1.2.2 Device Addressing

The four possible types of addresses that can be assigned to a Bluetooth unit are shown in Table 2.

Address	**Description**	
BD_ADDR	Bluetooth Device Address	A unique 48-bit device address allocated to each Bluetooth transceiver at manufacture time. It is divided into: • 24-bit LAP field • 16-bit NAP field • 8-bit UAP field
AM_ADDR	Active Member Address (BT unit MAC address)	A 3-bit number that is only valid as long as a slave is active on the channel. Up to 7 active members are thus supported.
PM_ADDR	Parked Member Address	8-bit number separating the parked slaves. Up to 254 parked members are thus supported.
AR_ADDR	Access Request Address	Used by a parked slave to determine the slave-to-master half slot in the access window it is allowed to send access request messages in. It is only valid as long as the slave is parked and is not necessarily unique.

Table 2: Bluetooth device addressing

2.1.1.2.3 Physical Links

The baseband handles two types of links: Synchronous Connection Oriented (SCO) and Asynchronous Connection-Less (ACL), compared in Table 3.

Attribute	**SCO**	**ACL**
Link type	Symmetric point-to-point	Point-to-multipoint
Slot usage	reserved	Per-slot basis
Transmission timing	Regular intervals / synchronous	Asynchronous
Data type	Voice (64 kB/s speech)	Data packets
Analogy	Circuit switched	Packet switched
Error correction	No retransmission	Packet retransmission
Number of links allowed	Up to 3	1

Table 3: Bluetooth Baseband Physical Links

2.1.1.2.4 Logical Channels

Bluetooth uses the following five logical channels to transfer different types of information:

1. Control Channel (LC)
2. Link Manager (LM)
3. Asynchronous User Information (UA)
4. Isochronous User Information (UI)
5. Synchronous User Information (US)

2.1.1.2.5 Packets

Table 4 shows the 13 different packet types used by the Bluetooth baseband layer.

Packet	Packet Description	Application	ACL	SCO	Bytes	Slots	CRC	FEC	kbps
ID	Device Access or Inquiry Access Code	Paging & Inquiring	●	●		1			
NULL	Channel Access Code & Packet Header	Flow Control	●	●		1			
POLL	Similar to NULL with Slave Response	Flow Control	●	●		1			
FHS	Frequency Hop Synchronisation	Piconet setup and hop synchronisation	●	●		1			
DM1	Data Medium Rate 1	Packet Data	●	●	18	1	16 bit	2/3	108.8
DH1	Data High Rate 1	Packet Data	●		28	1	16 bit	-	172.8
AUX1	Aux Data High Rate	Packet Data	●		30	1	-	-	
DM3	Data Medium Rate 1	Packet Data	●		123	3	16 bit	2/3	258.1
DH3	Data High Rate 3	Packet Data	●		185	3	16 bit	-	390.4
DM5	Data Medium Rate 1	Packet Data	●		226	5	16 bit	2/3	286.7
DH5	Data High Rate 5	Packet Data	●		341	5	16 bit	-	433.9
HV1	High Quality Voice 1	Voice Data		●	10	1	-	1/3	64
HV2	High Quality Voice 2	Voice Data		●	20	1	-	2/3	64
HV3	High Quality Voice 3	Voice Data		●	30	1	-	-	64
DV	Data Voice Combined Packet	Voice & Packet Data		●	10 18	1	- -	- 2/3	64

Table 4: Bluetooth Baseband Packet Types

Figure 3 shows the typical format of a Bluetooth baseband packet, containing an access code, header and payload entity, which is each described in Table 5.

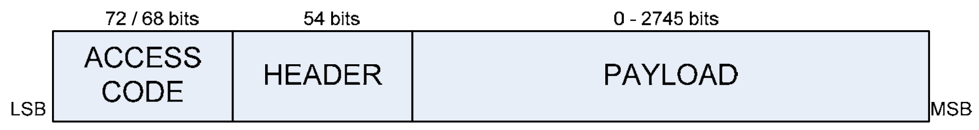

Figure 3: Bluetooth baseband packet format

Packet Entity	Description
Access Code	Used for timing synchronisation, offset compensation, paging and inquiry. Table 6 describes the three different types of access codes.
Header	Contains information for • packet acknowledgement • packet numbering (for out-of-order packet reordering) • flow control • slave address • header error checking
Payload	Contains either a voice field, data field, or both. For data fields, the payload will also contain a payload header.

Table 5: Baseband packet entities

Access Code	Description	
CAC	Channel Access Code	Unique piconet identifier
DAC	Device Access Code	Used for paging and paging responses
IAC	Inquiry Access Code	Used for inquiry and inquiry responses

Table 6: Baseband packet access code types

2.1.1.2.6 Channel Control

A Bluetooth controller operates in two major states: **Standby** and **Connection**, with a number of substates required to make a connection. An *inquiry procedure* is used to discover devices and a *paging procedure* is followed to establish an actual connection. When some details about a remote device is already known, only the *paging procedure* is required for the connection establishment. All the controller states are described in Table 7.

State	Description
Standby	This is the default low power state where only the native clock is running with no interaction to other devices.
Inquiry	A source device enters the inquiry state to discover which devices are in range by sending out inquiry packets.
Inquiry Scan	A destination unit in inquiry scan state will receive the inquiry packets from the source and enter inquiry response state.
Inquiry Response	A destination in inquiry response state will send an inquiry reply to the inquiring source.
Page	A source device longing to make a connection with a destination device (knowing its BT address and optionally its clock) sends it paging packets.
Page Scan	A destination unit in page scan state will receive the paging packets from the source and enter slave response state.
Slave Response (1)	A destination in slave response state will send a page reply to the paging source.
Master Response (1)	The source device (master of the link) sends an FHS packet to the destination.
Slave Response (2)	The destination sends it's second reply to the source.
Master Response (2)	Both the destination and source then switch to the source channel parameters.
Connection	This state involves a master and slave exchanging packets, starting with a POLL packet sent by the master to verify that the slave has switched to its timing and channel frequency hopping. In this state the device can take on one of the following four **connection modes**: Active, Sniff, Hold or Park

Inquiry Procedure
The inquiry procedure enables a device to discover which devices are in range, and determine the addresses and clocks for the devices (which is required to make a connection).

Page Procedure
The paging procedure is used to establish an actual connection between two devices typically after the inquiry procedure. Only the destination device address is required, but knowledge about its clock will accelerate the procedure. The source device carrying out the paging procedure automatically becomes the master of the connection.

Table 7: Channel Controller States and Connection Setup

An existing master and slave may also swap roles by applying the following two steps:

1. Perform a TDD switch followed by a piconet switch.
2. Transfer slaves from the old piconet to the new piconet. After acknowledging reception of the FHS packet, a unit will use the new piconet parameters, completing the switch.

Table 8 describes the connection modes a device can take on when in the connection state (also mentioned in Table 7).

Mode	Description	Power Usage
Active	The unit actively participates on the channel. As master, it schedules transmissions based on traffic demand or to keep slaves synchronised. As slave it listens to packets in the master-to-slave slot.	Most
Sniff	A slave device saves power by reducing the rate at which it listens to the piconet, with the interval being programmable.	↓
Hold	A slave device can be put into HOLD mode by the master, where only its internal timer is running. A slave device can also demand this from the master.	↓

Park	A slave device does not participate in piconet traffic, but is still synchronised although it has given up its MAC address. Occasionally it will re-synchronise and check on broadcast messages.	Least

Table 8: Connection Modes in the Connection State

2.1.1.2.7 Error Correction

Table 9 shows the three error correction schemes used in the baseband protocol:

Scheme	Description
1/3 rate FEC	Every bit is repeated three times for redundancy
2/3 rate FEC	A 10-bit code is encoded into a 15-bit code by using a generator polynomial
ARQ	DM and DH packets, and the data field of DV packets are retransmitted until either of the following conditions are met: • an acknowledgement is received. Fast, unnumbered, positive and negative acknowledgements are used by setting appropriate ARQN values. • a timeout is exceeded. The sender then flushes the packet and proceeds with the next.

Table 9: Baseband Error Correction Schemes

2.1.1.2.8 Flow Control

When data cannot be received (due to a full RX FIFO queue), flow control is implemented by transmitting a **stop** indication in the header of the return packet. Upon receiving, the transmitter freezes its FIFO queues. After congestion is cleared, the receiver sends a **go** packet to resume flow.

2.1.1.2.9 Synchronising

The piconet is synchronised by the system clock of the master. The slaves adapt their native clocks with a timing offset in order to match the master clock giving them an estimated clock value. The Bluetooth clocks should have the LSB ticking in units of 312.5 us, giving a clock rate of 3.2 kHz. A 20 us uncertainty window is allowed around the exact receive time in order for the access correlator to search for the correct channel access code and get synchronised with the transmitter. A slave in hold mode will correlate over a bigger uncertainty window. A slave in parked mode periodically wakes up to listen to beacons from the master and re-synchronises its clock offset.

2.1.1.2.10 Security

By using a unique public device address (BD_ADDR), two secret keys (authentication and encryption) and a random number generator, security at the baseband layer is obtained by:

- **Authentication**: a challenge is send to a remote device, which should send a response based on the challenge itself, it's BD_ADDR and a link key shared between the two devices.
- **Encryption**: after a device has been authenticated, data may be encrypted for further communication.

2.1.1.3 Link Manager Protocol (LMP)

The Link Manager Protocol (LMP) is used by the Link Managers (LM) on either side to set-up and control a link. An LM discovers and communicates with remote LM's by using the LMP and services from its underlying Link Controller (LC).

A number of Protocol Data Units (PDU's) makes up the Link Manager Protocol. The AM_ADDR in the packet header, determine the destination device. The header is one byte, with the PDU's always sent as single-slot packets.

LM PDU's are transported by DM1 packets. When an SCO link is present using HV1 packets and the content length is less than 9 bytes, DV packets are used.

The extensive list of Bluetooth PDU's will not be covered in this overview.

2.1.1.4 Host Controller Interface (HCI)

The Host Controller Interface provides a command interface to the Baseband Link Controller and Link Manager. It also provides access to the hardware status and control registers, delivering a uniform access method to the Baseband. The HCI exist across 3 sections with a functional part in each, described in Table 10.

HCI Functional Part	Bluetooth system Section	Description
HCI Firmware	Host Controller (e.g. Bluetooth hardware device)	Implements the HCI commands for the Bluetooth hardware by accessing: • Baseband commands • Link manager commands • Hardware status registers

		• Control registers • Event registers
HCI Driver	Host (e.g. software entity)	Used to receive and parse asynchronous HCI events.
Host Controller Transport Layer	Intermediate Layers (e.g. several layers between the host and host controller)	The several layers between the HCI driver and the host system via which communication is transported without intimate knowledge of the underlying data. Several different Host Controller Layers may be used, of which 3 have been defined initially for Bluetooth; • UART: provide Bluetooth HCI usage over a serial interface between two UARTs on the same PCB, where event and data packets can flow through the layer without being decoded. This UART Transport Layer assumes that the UART communication is free from line errors. • RS232: provide HCI usage over a single physical RS232 interface between the Host and the Host Controller, where event and data packets can flow through the layer without being decoded. • USB: Provide a Universal Serial Bus (USB) hardware interface to the Bluetooth hardware, using a specific class code to load the proper driver stack, regardless of vendor.

Table 10: HCI functional parts in their respective sections

The HCI provides the host with the ability to control the link layer capabilities via a uniform command method. These commands are described in more detail in Table 11.

HCI Command	Description
HCI-Specific Information Exchange	Provide the ability for the host to send HCI commands, ACL and SCO data to the Host Controller, and also receive HCI events, ACL and SCO data. The format of these commands, events and data exchange is specified by the data specification.
Link Control Commands	Provide the host with the ability to control the link layer connections (how piconets and scatternets are established and maintained), typically involving the Link Manager to exchange LMP commands.
Link Policy Commands	Used to affect the behaviour of the local and remote LM, providing it with methods to influence the piconet management according to adjustable policy parameters.
Host Controller & Baseband Commands	Provide access to various capabilities of the Bluetooth hardware, controlling the capabilities of the Host Controller, Link Manager and Baseband. The host device can use these commands to modify the behaviour of the local device.
Informational	Provide information about the Bluetooth device and the capabilities of

Parameters	the Host Controller, Link Manager and Baseband. These parameters are fixed by the manufacturer of the device.
Status Parameters	Provide information about the current state of the Host Controller, Link Manager and Baseband. The host device cannot modify any of these parameters, other than to reset certain specific parameters.
Testing Commands	Used to provide the ability to test various functionalities of hardware by providing the ability to arrange various conditions for testing.

Table 11: HCI Commands overview

A number of different **HCI events** are defined of which 32 has been implemented in the Bluetooth stack, ranging from *Inquiry Complete Event* to *Page Scan Repetition Mode Change Event*. Similarly a number of **error codes** has been defined of which 35 has been implemented, ranging from *Unknown HCI Command* to *LMP PDU Not Allowed*. All these events and error codes are detailed in the HCI Specification [4].

The Host implements **flow control** by managing the data buffers of the Host Controller, avoiding filling it up when a remote device is not responding.

2.1.1.5 Logical Link Control and Adaptation Protocol (L2CAP)

Logical Link Control and Adaptation Protocol (L2CAP) provides connection-oriented and connectionless data services to higher levels with protocol multiplexing, packet segmentation and reassembly, group abstractions and the conveying of quality of service information (described in Table 12). L2CAP permits higher level protocols and applications to transmit and receive L2CAP data packets up to 64 kilobytes in length, with the default MTU (Maximum Transmission Unit) set to 672 bytes. Of the two link types supported by the Baseband layer, only ACL links providing best effort traffic are supported by the L2CAP specification. L2CAP has no support for SCO links providing real-time voice traffic.

The *connection-oriented data channel* represents a connection between two devices, where a CID identifies each endpoint of the channel. The *connection-less data channels* restrict data flow to a single direction, used to support a channel 'group' where the CID on the source represents one or more remote devices (like broadcasting).

L2CAP implementations must provide the following operations:

- Data transfers between higher and lower protocol layers
- Support signalling commands between two L2CAP implementations
- Accepts certain lower layer events and generate events for upper layers.

Protocol Function	Description
Protocol Multiplexing	L2CAP must support multiplexing in order to distinguish between upper layer protocols (like Service Discovery Protocol, RFCOMM and Telephony Control).
Segmentation & Reassembly	Other than wired media, the data packets supported by the Baseband are limited in size, requiring segmentation of large L2CAP packets prior to Baseband transmission. Similarly these packets should be reassembled on the receiver side into a single large L2CAP packet following a simple integrity check. This function is described further in section 2.1.1.5.2.
Quality of Service	Upon establishing an L2CAP connection, information regarding quality of service (QoS) is exchanged between the two Bluetooth units. The L2CAP implementation must monitor the resources used by the protocol and ensure that QoS requirements are met.
Groups	The L2CAP group abstraction permits other protocol groups (implementing the concept of a group of addresses) to be mapped on to piconets. Without this, exposure to the Baseband and Link Manager would have been needed to manage groups efficiently.

Table 12: L2CAP Protocol Functions

Several services are offered by L2CAP in terms of service primitives and parameters:

- **Connection**: setup, configure, disconnect
- **Data**: read, write
- **Group**: create, close, add member, remove member, get membership
- **Information**: ping, get info, request a call-back at the occurrence of an event
- **Connection-less Traffic**: enable, disable

Configuration parameter options are provided to extend the ability of negotiating different connection requirements. It is transmitted in the form of information elements comprised of a type, length and data field.

2.1.1.5.1 L2CAP Channelling

The L2CAP layer is based around the concept of ‘channels’, with each end-point referred to by a *channel identifier* (CID), a local name representing the channel end-point on the device. The only restriction on CIDs is that it may not be reused for multiple simultaneous channels, and some CIDs are reserved:

- **Signalling channel** (CID 0x0001): used to create and establish connection-oriented data channels and negotiate changes in the channel characteristics. MTU commands take the form of Requests and Responses, and multiple commands may be sent in a single (L2CAP) packet. The BD_ADDR of the sending/requesting device must also be prevalent.

- **Uni-directional channel**: reserved for all incoming connectionless data traffic.

2.1.1.5.2 L2CAP Segmentation and Reassembly

Although L2CAP follows a channel based communications model, it is still packet-based. Segmentation and Reassembly (SAR) of packets are used to improve efficiency by supporting a maximum transmission unit (MTU) size larger than the largest Baseband packet. Larger packets used by upper layers (with a format that support adaptation to smaller physical frame sizes) are then segmented into chunks for transfer over Baseband packets. On the receiver side the L2CAP reassembles these chunks received from the HCI using information obtained from the HCI and packet header. All packet fields use Little Endian byte order.

2.1.1.6 Synchronous Audio Data

2.1.1.6.1 Synchronous Connection Oriented (SCO) Logical Link

The Synchronous Connection-Oriented (SCO) logical transport is a symmetric, point-to-point channel that can be implemented for transmitting a synchronous constant sized stream of data (typically encoded voice data) between two remote Bluetooth devices at a fixed rate. The SCO logical transport reserves slots on the physical Bluetooth channel and can therefore be considered as a circuit switched connection between the master and slave device, whereas the asynchronous ACL logical transport supported by L2CAP (see section 2.1.1.5) can be considered as a packet switched connection. SCO logical transports carry 64 kbps of information synchronised with the piconet clock. A device uses a combination of slot number, packet type and LT_ADDR (same as the default ACL logical transport) to distinguish between different transmissions on a singular SCO link. The single logical transport channel that these links are associated with, means that the packets will have a predefined fixed length and scheduling period and not contain an LLID (Logical Link Identification).

2.1.1.6.2 Extended Synchronous Connection Oriented (eSCO) Logical Link

The extended Synchronous Connection-Oriented (eSCO) logical transport is an asymmetric, point-to-point channel between two devices. eSCO supports a more flexible combination of packet types and selectable slot periods, allowing a wider range of synchronous bit rates to be supported. It also offers limited retransmission of packets.

2.1.1.7 Service Discover Protocol (SDP)

The Service Discovery Protocol (SDP) provides a means for applications to discover which services are provided by a linking Bluetooth device, and the characteristics of those services. The protocol addresses service discovery specifically for the Bluetooth environment with its highly dynamic nature. The protocol does not define methods for accessing the services provided (which in general are accessed using the other protocols defined in Bluetooth).

2.1.1.7.1 SDP Overview

A specific Service Discovery Protocol is needed for the unique characteristics of the Bluetooth environment, as available services may change dynamically based on the RF proximity of devices in motion. SDP is a simple protocol with minimal requirements on the underlying transport layers. It can function over a reliable or unreliable packet transport protocol. SDP uses a request response model where each transaction consists of one request Protocol Data Unit (PDU) and one response PDU. When SDP uses the L2CAP transport protocol, only a single SDP request to an SDP server may be outstanding at a given instant, providing a simple form of flow control.

Some requests may however require responses that are larger than what can fit in a single response PDU. To extend the response to more than a single response PDU, the SDP server will generate a *partial response* along with a *continuation state parameter*. This *continuation state parameter* can then be supplied by a client in a subsequent request to retrieve the next portion of the incomplete response.

In cases where a server determines that an error has occurred (a request is ill-formatted or an appropriate PDU response can be generated), it will respond with an error PDU (SDP_ErrorResponse).

An SDP PDU consists of a PDU header followed by PDU-specific parameters. The header consists of the following three fields:

- **PDU ID**: identifies the type of PDU.
- **Transaction ID**: identifies request PDU’s or match response PDU’s to a request.
- **Parameter Length**: specifies the length (in bytes) of all parameters contained in the PDU.

2.1.1.7.2 SDP Services

A service is an entity that can provide information, perform an action, or control a resource on behalf of another entity. A service may be implemented as software, hardware, or a combination thereof. Information about a service is stored in a single **service record** consisting of a list of **attributes**, each describing a single characteristic of the service. A database of these records is maintained by an **SDP server**. Some attribute definitions are common to all services, but service providers can also define their own service attributes. A service attribute comprises out of these two components:

- **Attribute ID**: a 16-bit unsigned integer hat distinguishes each service attribute from other service attributes within a service record. The ID also identifies semantics of the associated attribute.
- **Attribute Value**: a variable length element whose meaning is determined by its attribute ID and the service class of the service record in which it is contained. It consists of two parts:
    - **Header field**: The header field is composed of two parts:
        - **Type Descriptor**: represents the data element type using a 5-bit value.
        - **Size Descriptor**: represents the data element size using a 3-bit index followed by 0, 8, 16 or 32 bits.
    - **Data field**: a sequence of bytes of length specified by the size descriptor, and meaning specified by the type descriptor.

Each service is an instance of a **service class**, comprising:

- a **service class ID**: represented by a 128-bit universally unique identifier (UUID) contained as an attribute in the ServiceClassIDList. Examples of UUIDs include:
    - PNP: 00001200-0000-1000-8000-00805F9B34FB
    - HID: 00001124-0000-1000-8000-00805F9B34FB
    - Headset: 00001108-0000-1000-8000-00805F9B34FB
- a **service class definition**: provides definitions of all attributes contained in its service record instances.
- **attribute definitions**: specifies the numeric value of the attribute ID, the intended use of the attribute value, and its format.

A service record contains attributes that are specific to a service class, as well as universal attributes that are common to all services.

2.1.1.7.3 Service Discovery

SDP allows Bluetooth devices to discover what services other Bluetooth devices can offer, by either:

- **Searching –** looking for a specific service: allows a client to retrieve particular service record handles based on their attribute values. This capability is not available for arbitrary attributes, but only attributes which values are UUIDs. The desired service is located by using a service search pattern (a list of UUIDs with matching service records).
- **Browsing –** looking at all the services actually on offer: allows a client to browse through all the services on offer by an SDP server. A **BrowseGroupList** attribute, shared by all service classes contains a list of UUIDs each representing a group with which a service may be associated. All services that may be browsed at the root level are made members of the root browse group by having its UUID as a value within the BrowseGroupList attribute.

2.1.1.8 RFCOMM

The emulation of (RS232) serial ports over the L2CAP protocol is provided by the RFCOMM protocol, based on a subset of the ETSI standard TS 07.10.

2.1.1.8.1 Overview

RFCOMM emulates the 9 control signals of an RS-232 interface, shown in Table 13.

Pin	Signal	Description
102	GND	Common
103	TD	Transmit Data
104	RD	Receive Data
105	RTS	Request to Send
106	CTS	Clear to Send
107	DSR	Data Set Ready
108	DTR	Data Terminal Ready
109	CD	Data Carrier Detect
125	RI	Ring Indicator

Table 13: RS232 Control Signals emulated by RFCOMM

An RFCOMM session is identified by the BD_ADDR of the two endpoints. At any time, there must be at most one RFCOMM session between any pair of devices. RFCOMM supports up to 60 simultaneous connections over a single session, but the actual number that can be used is device/implementation dependent. A Data Link Connection Identifier (DLCI) identifies an ongoing connection between a client and a server application, with values ranging from 2 to 61. The DLCI value space is divided between two communicating devices (using the concept of RFOMM server channels and a direction bit) to accommodate both client and server applications residing on both sides of an RFCOMM session. For multiple emulated serial ports with endpoints in different Bluetooth devices, multiple TS 07.10 multiplexer sessions must be run.

RFCOMM must support two basic device types, although the protocol does not make a pertinent distinction between the two:

- **Type 1 Devices**: communication endpoints (e.g. computers and printers)
- **Type 2 Devices**: communication segments (e.g. modems)

Serial ports for these device types are emulated by RFCOMM using the following entities:

- **Port Emulation Entity**: maps a system specific communications interface (API) to the RFCOMM services.
- **Port Proxy Entity**: relays data from the RFCOMM to an external RS232 interface linked to a DCE, with the communications parameters set according to received RPN commands.

RFCOMM provides **reliability** by requiring remote responses for some frame types (SABM, DISC and UIH frames with multiplexer control commands). Data frames however do not require responses on the RFCOMM level, and therefore L2CAP channels with maximum reliability should be used (ensuring all frames are delivered in order without duplicates). If L2CAP fails to provide this, RFCOMM should handle the expected Link Loss notification.

RFCOMM itself do not suffer from latency delays incurred by low power modes and consequently the specification does not state any latency requirements. Latency requirements by an application should be aggregated and conveyed to L2CAP by the RFCOMM implementation. This information may be used by lower layers to decide which low power mode (see Table 8) to use when the L2CAP channel remains idle for a certain amount of time.

2.1.1.8.2 TS 07.10 adaptations

The Bluetooth RFCOMM specification requires some adaptations of the ETSI standard:

- **Media Adaptation**: RFCOMM does not use the opening and closing flags in the 07.10 basic option frame. Only the fields between these two flags are used in an RFCOMM frame, which fits exactly into one L2CAP frame.

- **Multiplexer Startup & Closedown Procedure**: The 07.10 opening and closing flags are not supported in RFCOMM, but the following procedures instead:
    1. *Startup*: A device opening the first serial connection (DLC) is responsible for establishing the multiplexer control channel, by:
        - establishing an L2CAP channel to the peer RFCOMM entity using L2CAP service primitives,
        - start the RFCOMM multiplexer (by sending an SABM command on DLCI 0 and waiting for a UA response)
    2. *Closedown*: A device closing the last connection (DLC) on a particular session is responsible for closing the multiplexer (by closing the corresponding L2CAP channel).
    3. *Link Loss Handling*: On the event of receiving an L2CAP Link Loss Notification, the RFCOMM is responsible for sending a connection loss notification to the port emulation entity for each active DLC, freeing all resources associated with an RFCOMM session.

- **DLCI Allocation with RFCOMM Server Channel**: To enable both client and server applications to reside on both sides of an RFCOMM session (with connections being established independent of each other), the DLCI value space is divided between the two communicating devices using the concepts of RFCOMM server channels and a direction bit (D=1 for the initiating device). The RFCOMM server channel is a subset of the bits in the DLCI part of the address field in the TS 07.10 frame. A registering server application is assigned a Server Channel Number in the range 1...30. This number, together with the direction bit, is used to derive the DLCI used for all packets exchanged between two endpoints.

- **Multiplexer Control Commands**: shows the supported commands for controlling the multiplexer (note that some of these commands may be exchanged on the control channel (DLCI 0) before the corresponding DLC has been established.

Control Command	ID	Implementation
Remote Port Negotiation	RPN	Used before a new DLC is opened or when the port settings change
Remote Line Status	RLS	Used for indication of remote port line status
DLC Parameter Negotiation	PN	Used before first DLC creation, with the initiator trying to turn on the use of credit base flow control.

Table 14: Multiplexer Control Commands

2.1.1.8.3 Flow Control Methods

Different flow control methods are used when emulating a serial port over RFCOMM, all described in Table 15.

Flow Control Method	**Description**
L2CAP Flow Control	L2CAP uses the flow control mechanism provided by the Link Manager in the Baseband (see section 2.1.1.3). Flow Control from the RFCOMM layer to L2CAP is implementation specific.
Wired Serial Port Flow Control	The following methods may be used in one direction, or on both sides of the wired link: • Software Flow Control: use characters such as XON/XOFF • Hardware Flow Control: use circuits such as RT/CTS or DTR/DSR
RFCOMM Flow Control	RFCOMM provides two flow control mechanisms: • Global: flow control commands operating on the aggregate data flow between two RFCOMM entities (all DLCIs) • Individual: the Modem Status command operate on an individual DLCI
Port Emulation Entity: Serial Flow Control	• Port drivers, interfacing an application to RFCOMM, will need to provide flow control mechanisms as specified by the API they are emulating. Since RFCOMM already has its own flow control, the port driver does not need to perform flow control using the methods requested by the application.
Credit Based Flow Control	Credit based flow control provides flow control on a per-DLC basis. Both sides of an RFCOMM session will know the buffer sizes (credits) of each DLC on the connecting device. A sending device must stop transmission and wait for further credits when the destination DLC's credit counter reaches zero. This mechanism does not apply for DLCI 0 or non-IH frames, and is only supported by Bluetooth version following 1.0B.

Table 15: Flow Control Methods used when emulating serial ports over RFCOMM

2.1.1.9 Application Layer

Bluetooth application software can open communication channels over the Bluetooth subsystem in various different ways. This application side of Bluetooth is explored in the developed implementation for analysing the data transmitted over a Bluetooth connection, described in section 3.2.2 and Addendum B.

2.1.2 *Bluetooth Profiles*

The Bluetooth profiles describe how the different parts of the protocol stack can be applied to fulfil a desired function for a Bluetooth device. A number of user scenarios with Bluetooth as radio medium are described. Mandatory options and parameter ranges in each protocol layer are defined for each profile, describing a vertical slice through the stack. The concept of profiles is used to decrease interoperability risks between manufacturers.

Table 16 shows a quick overview of some Bluetooth profiles commonly encountered in practice.

Bluetooth Profile		**Description**
GAP	Generic Access Profile	The core on which all other profiles are based and defines the generic procedures related to discovery of Bluetooth devices and link management aspects of connecting to Bluetooth devices.
DUNP	Dial-up Networking Profile	Requirements for Bluetooth devices necessary to support the Dial-up networking use case. Essentially the Headset profile defines the protocols and procedures that shall be used by devices implementing the usage model called ‘Internet Bridge', e.g. modems and cellular phones.
SPP	Serial Port Profile	For setting up emulated serial cable connections using RFCOMM between two peer devices, expressed in terms of services provided to applications and by defining the features and procedures that are required for interoperability between Bluetooth devices.
LAP	LAN Access Profile	Defines how Bluetooth-enabled devices can access the services of a LAN using PPP. Secondly, this profile shows how the same PPP mechanisms are used to form a network consisting of two Bluetooth-enabled devices, using PPP over RFCOMM.
SDAP	Service Discovery Application Profile	Defines the features and procedures for a Bluetooth application to discover services registered in remote Bluetooth devices and retrieve information on it.
CTP	Cordless Telephony Profile	Defines the requirements for interoperability between different units active in the ‘3-in-1 phone’ use case (a solution for providing an extra mode of operation to cellular phones) using Bluetooth as a short-range bearer for accessing fixed network telephony services via a base station.
IP	Intercom Profile	Requirements for devices supporting the intercom functionality within the 3-in-1 phone use case, expressed in terms of end-user services and defining the features and procedures that are required for interoperability between devices.
HS	Headset Profile	Devices necessary to support the headset use case, essentially defining the protocols and procedures that shall be used by devices implementing the usage model called ‘Ultimate Headset’. The most common examples of such

		devices are headsets, personal computers and cellular phones.
FP	Fax Profile	Defines the requirements to support the Fax use case, i.e. the protocols and procedures that shall be used by devices implementing the fax part of the usage model called ‘Data Access Points, Wide Area Networks’. A Bluetooth cellular phone or modem may be used by a computer as a wireless fax modem to send or receive a fax message.
OBEX	Generic Object Exchange Profile	Defines the Bluetooth requirements necessary for Synchronisation, File Transfer, or Object Push model. Essentially, the purpose of this profile is to work as a generic profile document for all application profiles using the OBEX protocol.
OPP	Object Push Profile	The object push usage model makes use of the underlying Generic Object Exchange Profile (OBEX) to define the interoperability requirements for the protocols needed by applications. Typical scenarios covered by this profile are Object Push, Business Card Pull & Business Card Exchange, all of which involve the pushing/pulling of data objects between Bluetooth devices.
FTP	File Transfer Profile	This usage model also makes use of the underlying Generic Object Exchange Profile (OBEX) to define the interoperability requirements for the protocols needed by applications. Typical scenarios covered by this profile involve a Bluetooth device browsing, transferring and manipulating objects on/with another Bluetooth device.
SP	Synchronisation Profile	Also making use of OBEX to define the interoperability requirements for the protocols needed by applications, typical scenarios covered by this profile involves a computer instructing a mobile phone or PDA to exchange PIM data, or automatically starting synchronisation when 2 Bluetooth devices come within range.

Table 16: Bluetooth Profiles Description

2.2 ULTRA WIDEBAND

Ultra Wideband is a relatively new radio technology designed for wireless data transfers of high bandwidth and low power requirements. Ultra Wideband was traditionally accepted as pulse radio, but the Federal Communications Commission (FCC) now defines UWB in terms of a transmission from an antenna for which the emitted signal bandwidth exceeds the lesser of 500 MHz or 20% of the centre frequency. The signal spreading can be achieved by using either a pulse-based system which instantaneously occupies the complete allowed bandwidth, or an aggregation of narrow band carriers. Figure 4 gives an illustration of the UWB spectrum range [11].

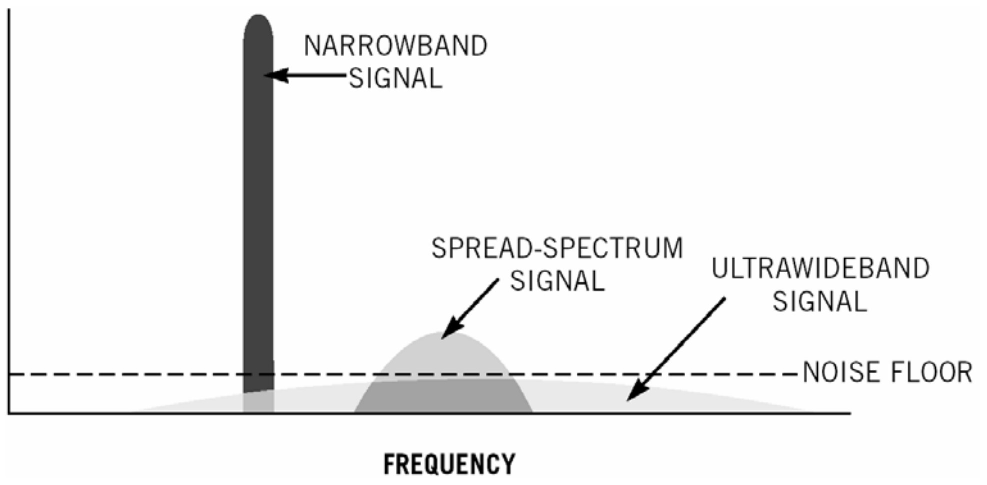

Figure 4: UWB signals in the spectrum range [11]

The lack of a standard has so far been the single biggest shortfall to the development of a sustainable UWB market [8]. Backed by a number of companies, the Multiband-OFDM Alliance (MBOA) was formed to promote an OFDM form of UWB within the IEEE802.15.3 group. On the other hand Motorola and XtremeSpectrum were opting for the single pulse CDMA approach with a notch around 5 GHz, but they were outnumbered by the MBOA now known as the WiMedia Alliance[1] [6], [9]. In March 2009 the WiMedia Alliance announced it is entering into technology transfer agreements of all current and future specifications to the Bluetooth Special Interest Group (SIG), Wireless USB Promoter Group and the USB Implementers Forum. After the successful completion of the technology transfer, the WiMedia Alliance will cease operations.

Because UWB covers spectrum above 3 GHz, it will not penetrate walls easily, as experience with Bluetooth, operating at only 2.4 GHz, has shown. The speed at close range

[1] WiMedia Alliance is a non-profit organization supported by more than 200 member corporations and research institutions, http://www.wimedia.org

proposed by MBOA is close to USB2 and FireWire, and therefore it is seen to become a lower-power much faster alternative to Bluetooth or WLAN. UWB's minimum speed of 110 Mbps at 10 meters is a 100 times faster than Bluetooth, and it can conceivably hit top speeds of 480 Mbps at 1 meter while using an average radiated power of 200 microwatts. This and the fact that consumer demand for connectivity among devices remains strong and will continue to grow stands UWB to capture technology market share over the next several years in a variety of applications, especially the distribution of wireless video, audio and data, in home, consumer networking (integration into desktops, laptops and PDA's for personal area networks), vehicle and locating devices [12], [13], [14].

2.2.1 ECMA UWB Standards

After failure of the FCC and IEEE to suitably standardise a universally acceptable UWB protocol, ECMA (European Computer Manufacturers Association) was engaged by the WiMedia Alliance to set a standard for the MB-OFDM. In the end two standards were released: the ECMA-368 for defining the physical (PHY) and Medium Access Control (MAC) layers of a system running in 3.1 up to 10.6 GHz, as well as ECMA-369, which describes the communications between the PHY and MAC.

2.2.1.1 ECMA-368 High Rate UWB PHY and MAC Standards

The ECMA-368 [26] standard, from hereon only denoted as UWB, specifies the popular WiMedia Ultra Wideband physical and Medium Access Control sub layers for a high-speed short range wireless network, supporting data rates of up to 480 Mb/s. The standard only defines a PHY and MAC layer to act as transmission sub layers for various different higher level protocols, like Wireless USB, Internet Protocol (IP) in the form of WiNet, and of course Bluetooth, as depicted in Figure 5.

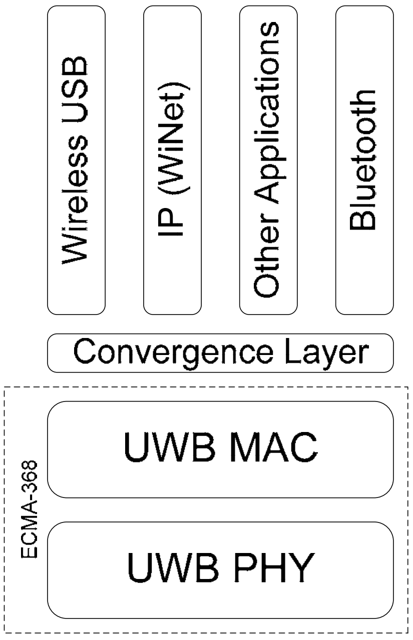

Figure 5: ECMA-368 UWB Protocol Stack

2.2.1.1.1 UWB PHY Specification

The standard divides the unlicensed 3.1GHz – 10.6 GHz frequency spectrum into 14 bands of 528 MHz, grouped into four band groups of 3 each and a fifth band group of 2 bands. For each band a total of 110 sub-carriers (100 data carriers and 10 guard carriers) and 12 pilot sub-carriers are used in a MB-OFDM scheme. Frequency domain spreading, time-domain spreading, and forward error correction (FEC) with rates 1/3, 1/2, 5/8 and 3/4 are used to vary the data rates between 53.3 Mb/s, 80 Mb/s, 106.7 Mb/s, 160 Mb/s, 200 Mb/s, 320 Mb/s, 400 Mb/s and 480 Mb/s.

The coded data is then spread over the frequency by either interleaving it over three bands with Time Frequency Interleaving (TFI) or transmitting it over a single band using Fixed Frequency Interleaving (FFI). The maximum allowed power emission is -41dBm/MHz, with a nominal value of -10dBm per band.

The first four band groups support seven channels per band, of which four uses TFI and three FFI. The fifth band group supports two FFI channels, bringing the total to 30 channels available for the PHY.

The PHY contains three functional entities: the Physical Medium Dependent (PMD) function, the PHY convergence function / Physical Layer Convergence Protocol (PLCP) sublayer, and the layer management function provided by the Physical Layer Management

Entity (PLME). The PHY service is provided to the MAC through the PHY service primitives.

2.2.1.1.1.1 PMD Sublayer

The Physical Medium Dependent sublayer provides a means to send and receive data between two or more stations.

2.2.1.1.1.2 PLCP Sublayer

In order to allow the MAC to operate with minimum dependence on the PMD sublayer, the PHY convergence sublayer is defined. This function simplifies the PHY service interface to the MAC services.

2.2.1.1.1.3 PLME Sublayer

The Physical Layer Management Entity performs management of the local PHY functions in conjunction with the MAC management entity.

2.2.1.1.1.4 Signal Description

A mathematical representation of the transmitted RF signal can be written in terms of the complex baseband signal as follows:

$$\mathrm{s}_{RF}(t) = \mathrm{Re}\left\{ \sum_{n=0}^{N_{packet}-1} s_n(t - nT_{SYM}) \exp(j2\pi f_c(q(n))t) \right\}, \qquad (1)$$

where $Re(\cdot)$ represents the real part of the signal, T_{SYM} is the symbol length, N_{packet} is the number of symbols in the packet, $f_c(m)$ is the centre frequency for the m^{th} frequency band, $q(n)$ is a function that maps the n^{th} symbol to the appropriate frequency band, and $s_n(t)$ is the complex baseband signal representation for the n^{th} symbol, which must satisfy the following property: $s_n(t) = 0$ for $t \notin [0, T_{SYM})$. The exact structure of the n^{th} symbol depends on its location within the packet:

$$\mathrm{s}_n(t) = \begin{cases} s_{sync,n}(t) & 0 \le n < N_{sync} \\ s_{hdr,n-N_{sync}}(t) & N_{sync} \le n < N_{sync} + N_{hdr} \\ s_{frame,n-N_{sync}-N_{hdr}}(t) & N_{sync} + N_{hdr} \le n < N_{packet} \end{cases}, \qquad (2)$$

where $s_{sync,n}(t)$ describes the n^{th} symbol of the preamble, $s_{hdr,n}(t)$ describes the n^{th} symbol of the header, $s_{frame,n}(t)$ describes the n^{th} symbol of the PSDU, N_{sync} is the number of symbols in the preamble, N_{hdr} is the number of symbols contained in the header, and $N_{packet} = N_{frame} + N_{sync} + N_{hdr}$ is the number of symbols in the payload.

2.2.1.1.1.5 Frame Structure

Figure 6 shows the structure of frames transmitted by the PHY. There are two types of preamble: Standard and burst. The PLCP header including MAC and PHY headers are protected by a header check sequence (HCS). The Frame Payload is followed by its frame check sequence (FCS).

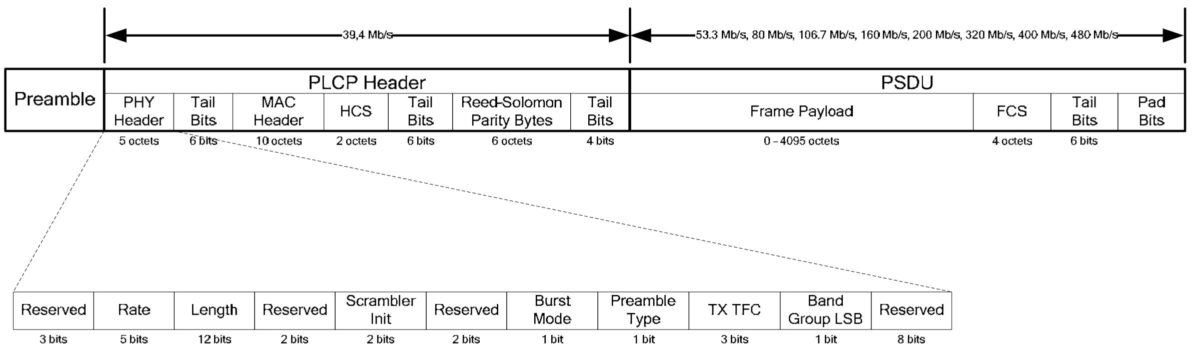

Figure 6: PHY frame structure

Frames are transmitted by the PHY from the source device and delivered to the destination device in identical bit order. The start of a frame refers to the leading edge of the first symbol of the PHY frame at the local antenna and the end of a frame refers to the trailing edge of the last symbol of the PHY frame. Frame transmission and reception are supported by the exchange of parameters between the MAC sublayer and the PHY layer. These parameters allow the MAC sublayer to control, and be informed of, the frame transmission mode, the frame payload data rate and length, the frame preamble, the PHY channel and other PHY-related parameters. In single frame transmission, the MAC sublayer has full control of frame timing. In burst mode transmission, the MAC sublayer has control of the first frame timing and the PHY provides accurate timing for the remaining frames in the burst.

2.2.1.1.2 UWB MAC Specification

The architecture of this MAC service is fully distributed. All devices provide all required MAC functions and optional functions as determined by the application. No device acts as a central coordinator.

2.2.1.1.2.1 Medium Access Schemes

The MAC uses a combination of Carrier Sense Multiple Access (CSMA) and Time Division Multiple Access (TDMA) to provide prioritised schemes for isochronous and asynchronous data transfers. For TDMA access, a Distributed Reservation Protocol (DRP) is used, and for CSMA, a Prioritised Contention Access (PCA). Table 17 shows the MAC sublayer prioritised medium access schemes used for data transfer, which together with other policies ensures equitable access of the bandwidth.

Traffic type	**Medium Access**	**Protocol**
Isochronous	Time Division Multiple Access (TDMA)	Distributed Reservation Protocol (DRP)
Asynchronous	Carrier Sense Multiple Access (CSMA)	Prioritised Contention Access (PCA)

Table 17: MAC Sublayer Prioritised Medium Access Schemes

2.2.1.1.2.2 MAC Addressing

The MAC addressing scheme includes unicast, multicast and broadcast values. Individual MAC sublayers are addressed via a EUI-48 and are associated with a volatile abbreviated address of 16 bits called a DevAddr, which is unique within the device’s extended beacon group.

2.2.1.1.2.3 Device Discovery and Channel Selection

Device discovery within radio range and network organisation is achieved by the exchange of periodic beacon frames, which also carry reservation and scheduling information for accessing the medium.

When a device is enabled, one or more channels are scanned for a beacon. If one is detected, the device synchronises its beacon period (BP) with it and uses this channel for exchanging data with members in its beacon group. If no beacons are detected in the channel, the device creates its own BP by sending a beacon. No transmissions other than beacons are permitted during the BP. To combat interference, each device has the capability to dynamically change the channel in which it operates without disrupting existing connections.

Each device protects its and its neighbours' BPs for exclusive use of the beacon protocol. No transmissions other than beacons are attempted during the BP of any device.

2.2.1.1.2.4 Logical groups

The MAC protocol is specified with respect to an individual device and its neighborhood forming a logical group to facilitate contention free frame exchange. These logical groups are categorised as either a beacon group or an extended beacon group. MAC protocol algorithms attempt to ensure that no member of the extended beacon group transmits a beacon frame at the same time as the device. Information included in beacon frames facilitates contention-free frame exchanges by ensuring that a device does not transmit frames while a neighbour is transmitting or receiving frames. To permit correct frame reception, MAC protocol algorithms attempt to ensure that a device's DevAddr is unique within the device's extended beacon group.

2.2.1.1.2.5 Timing and Asynchronous Data Transfer

Neighbouring devices exchange data frames on a TDMA basis by use of a timing structure called a superframe of 65 536 μs, depicted in Figure 7. The superframe is composed of 256 medium access slots (MASs) of 256 μs each. Each superframe starts with a BP at the beacon period start time (BPST).

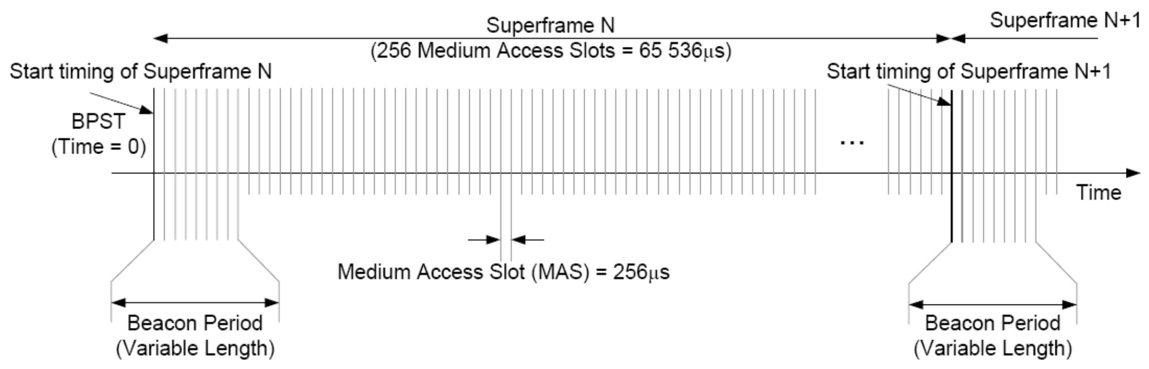
Superframe N
(256 Medium Access Slots = 65 536μs)
Superframe N+1
Start timing of Superframe N
Start timing of Superframe N+1
BPST
(Time = 0)
...
Time
Medium Access Slot (MAS) = 256μs
Beacon Period
(Variable Length)
Beacon Period
(Variable Length)

Figure 7: MAC superframe structure [26]

Data is passed between the MAC sublayer and its client in MAC Service Data Units (MSDUs) qualified by certain parameters. MSDUs are transported between devices in data frames. To reduce the frame error rate of a marginal link, data frames can be fragmented and reassembled. Fragments are numbered with an MSDU sequence number and a fragment number.

2.2.1.1.2.6 Reliability

Reliability is provided by the use of one of three acknowledgement policies; No-ACK, Imm-ACK or B-ACK.

2.2.1.1.2.7 Security

Security is provided by the protocol in the form of data encryption, message integrity and replay attack protection. Secure frames may be used by data frames and selected command and control frames. A 4-way handshake mechanism with pair-wise temporal keys (TPKs) based on a shared master key is used for unicast authentication, whilst multicast and broadcast frames make use of group temporal keys (GTKs). Payload encryption and message integrity codes (MICs) are generated by 128-bit symmetrical temporal keys based on AES-128.

2.2.1.1.2.8 Service Discovery

Information discovery are supported by broadcasting Information Elements (IEs) in beacon frames or requesting it in probe commands.

2.2.1.1.2.9 Power Management

The standard provides two power management modes, an active mode in which devices transmit in every superframe, and a hibernation mode where no data is transmitted for multiple superframes.

2.2.1.1.2.10 MUX sublayer

The standard also provides a MUX sublayer for routing MSDUs to and from different coexisting higher layer protocols.

2.2.1.2 ECMA-369 MAC-PHY Interface Standard for ECMA-368

The ECMA-369 [27] standard specifies the interface between implementations of the PHY and MAC as specified in ECMA-368 [26]. Figure 8 depicts the hardware interface between the MAC and the PHY.

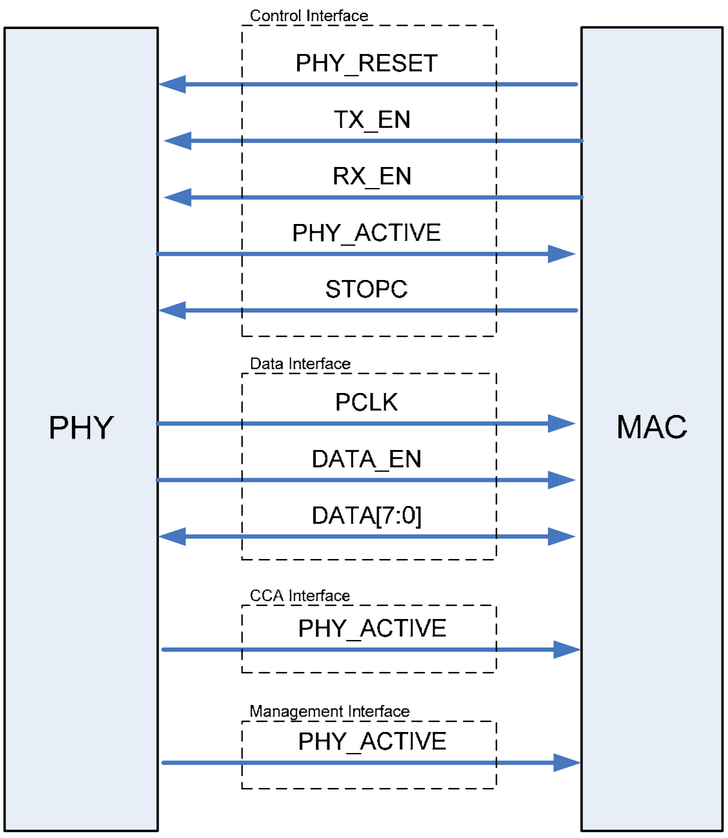

Figure 8: PHY – MAC Interface Signals

The PHY – MAC interface consists of the following sub-interfaces:

- *Data Interface* including an 8-bit data bus, used to transfer data to and from the MAC. This interface operates differently depending on the state of the PHY.
- *Control Interface* used by the MAC to control the operating state of the PHY and by the PHY to indicate TX/RX status to the MAC
- *Clear Channel Assessment (CCA) Interface* used for Clear Channel Assessment status indication.
- *Management Interface* used to access the PHY registers.

Interface functioning and frame structures used to operate this interface is described in detail in the ECMA-369 specification [27].

2.3 BLUETOOTH MERGENCE WITH UWB

The huge number of Bluetooth enabled products and Bluetooth enabled chipsets currently makes it the short-range wireless market leader, although it was predicted to transfer this position to UWB whilst coexisting with it in the market during the coming years. On the long run UWB is expected to take hold in multimedia applications that need higher data rates, whereas Bluetooth will fit in better with low speed, low data cable replacement in mobile terminals [10]. Analysts are predicting strong growth for UWB, with a possible 30 million chipsets shipped in 2008 at 12$ a piece, whilst the cost of adding Bluetooth should be well under $5 at this time. Power consumption is however likely to be a bigger consideration than chipset cost in determining whether high-speed, close-range UWB has a solid future [6], [8]. UWB eliminates the technology as the limiting factor in wireless networking solutions, for it provides enough throughput with low power and anticipated low cost [7].

There exist some similarities between UWB and Bluetooth as wireless communication protocols. Both operate over a short range (1 – 10 m) with minimal power requirements. Although the two standards make use of different unlicensed frequency bands, it has been shown that under extreme interference conditions UWB devices may have an impact on Bluetooth networks [28]. Both UWB and Bluetooth protocols have a layered architecture. This clustering of functionality eased our aim of imposing a clean break within the Bluetooth stack to merge with the lower-level UWB stack.

The HCI layer in Bluetooth provides a clear separation between upper layer Bluetooth protocol stacks, commonly implemented in software, and lower device dependant layers, commonly implemented in hardware. The interface consists of commands and events which can easily be imitated by a UWB Convergence layer [29].

A whole number of differences between the two protocol stacks need also be considered. Unlike Bluetooth’s master-slave scheme, the UWB MAC services are fully distributed with all devices providing the required functions and no device acting as central coordinator, enabling UWB devices to communicate directly with each other, and not only react to a single master device. Inherently there are also differences in the device discovery and connection setup process applied by the two standards. Whilst Bluetooth make use of page scanning, UWB uses beacons in a beacon period. In Bluetooth’s piconet with single master scheme, TDMA based channel access is also handled differently than with UWB’s prioritised medium access schemes based on TDMA and CSMA. With Bluetooth’s multiple piconet network topology all devices in range are synchronised and visible and don’t require explicit scanning to be discovered.

Some Bluetooth functionality can not be realistically implemented on top of a UWB physical-layer, for some of the higher level Bluetooth concepts rely on the characteristics of the Bluetooth lower physical-layer for functioning, e.g. the power saving link policy of Bluetooth uses the concept of “sniff” and “park” modes which are based on the time slot nature of the Bluetooth physical-layer [15]. An approach to making this selection would be to go through the Bluetooth Host Controller Interface (HCI) functional specification and check each Bluetooth command for viability in terms of the amount of glue layer translation functionality required for implementation over a UWB physical-layer.

It has also been recommended that stop-and-wait flow control in Bluetooth’s L2CAP layer is disabled when implementing UWB as physical transmission layer, for receiving UWB devices will not be able to acknowledge packets before its own DRP reservation [29].

CHAPTER 3: DESIGN AND IMPLEMENTATION

3.1 SYSTEM OVERVIEW

Despite various efforts at the time of starting this project, affordable Ultra Wideband hardware could just not be obtained due to manufacturers not having a research program in place and concerned only with high volume commercial prospects. To compromise, a Linux UWB stack [23] was applied to simulate UWB data flow by modifying the driver source code to rather send the physical layer data over an existing gigabit Ethernet connection, allowing for upper layer UWB MAC operation to still be performed on the UWB data, with the shortfall of realistically imitating UWB radio operations. Section 3.1.1 provides a description on how this simulation was implemented.

Due to the incorporation of UWB as physical layer in the Wireless Universal Serial Bus (WUSB) specification, affordable commercial UWB hardware has become readily available in the form of WUSB dongles as of 2008. Two of these dongles were acquired and their drivers modified to enable UWB communication between them, and applied as a real UWB physical link for the BToUWB system. This system is described is section 3.1.2.

3.1.1 Bluetooth over UWB Simulation

UWB offer speeds up to 480 Mbps. To simulate this UWB connection between two systems, the official Linux UWB driver were modified to channel its data over an existing gigabit Ethernet link between the systems. Figure 9 gives an overview of the implemented system design, followed by a more detailed look at its operation.

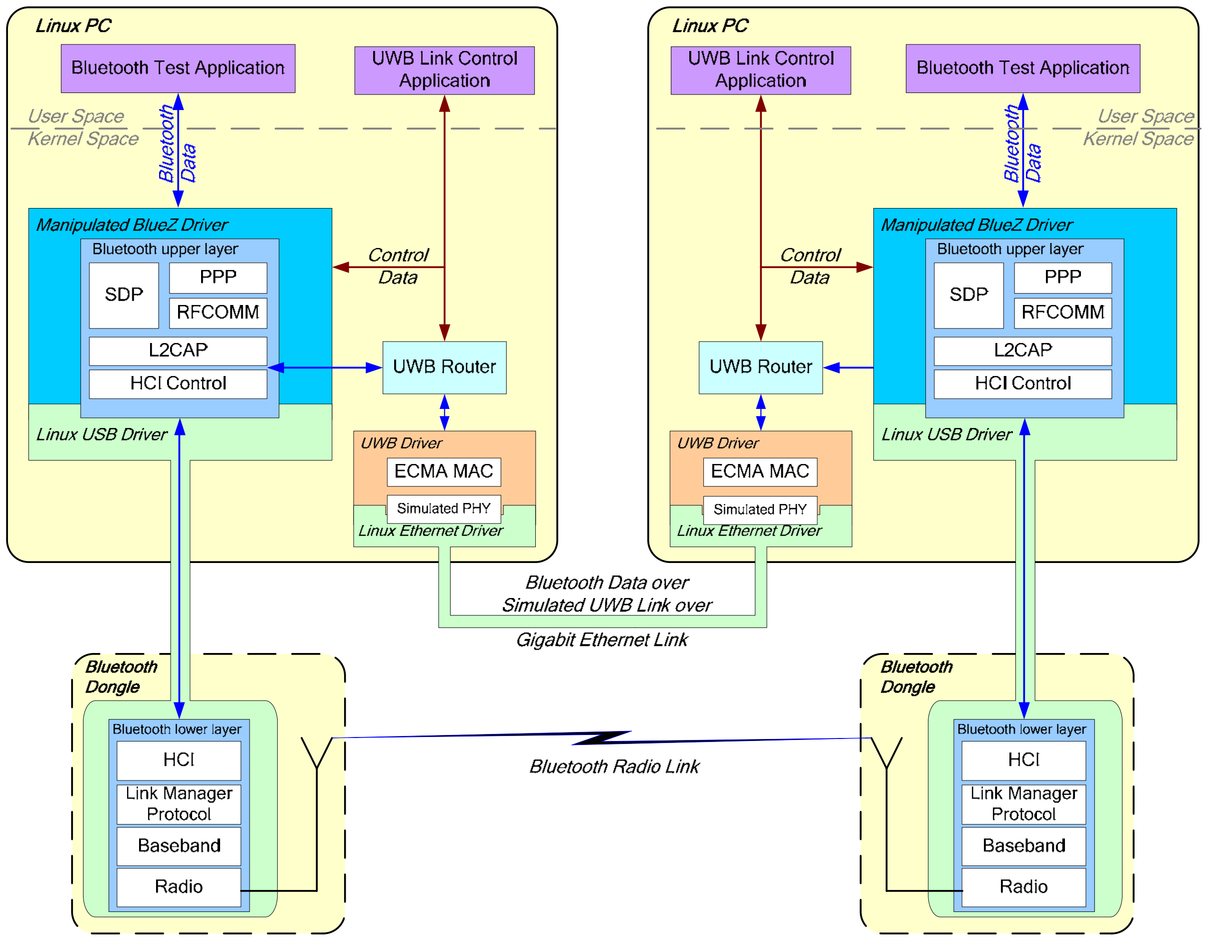

Figure 9: Implemented system diagram of Bluetooth over UWB simulation

To simulate Bluetooth dataflow over a UWB connection, a fully compliant Bluetooth connection were configured between two Bluetooth enabled computers under the open source Linux operating system [33]. The BlueZ stack used for this is described in section 3.2.1.

By modifying the open source BlueZ files the HCI commands sent to the HCI sublayer by upper layer L2CAP and SCO implementations were hijacked and routed to a *UWB Router* implementation for transmission over a UWB subsytem. Similarly lower level HCI events were spoofed to the L2CAP and SCO layers by the *UWB Router* implementation upon receiving packets from the UWB subsystem. More detail on this routing system is given in section 3.2.5.

To generate valid Bluetooth data for massaging the links, a Bluetooth analysis program has been developed according to proposed Bluetooth user level methods [30]. Section 3.2.2

provides details on how to develop user level Bluetooth programs, followed by a detailed description of this analysis program.

A *UWB Link Control* user level application was also developed to control the data routing performed by the routing subsystem. This application is also described further in section 3.2.5.

Prior to routing and analysing, all drivers need to be loaded and both the Bluetooth and simulated UWB links set up, by using the utilities listed in Addendums A.

3.1.2 Bluetooth over UWB System Implementation

As mentioned earlier in this chapter, UWB enabled hardware finally became affordably accessible in 2008 in the form of Wireless USB dongles. Two IO Gear GUWA100U Wireless USB Host Adapter dongles [36] were imported from the United States. These dongles, shown in Figure 10, were the first commercially available UWB hardware containing the Intel Wireless UWB Link 1480 Ultra Wideband MAC [24], which is supported by the open source Linux UWB driver, described in section 3.2.3.

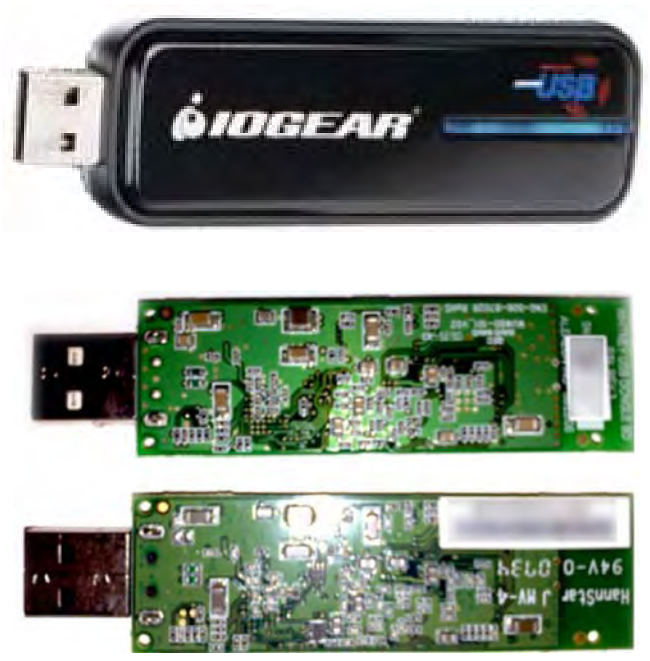

Figure 10: IO Gear GUWA100U Wireless USB Host Adapter [36]

From a systems point of view, an HWA itself consists of two main interfaces:

- Radio control (RC): this implements an interface to the Ultra Wideband radio controller.

- Host Controller (HC): a USB controller that is connected via USB.
- WiNet: IP over UWB that looks like a network interface.

To instantiate a Bluetooth over physical UWB system, a Bluetooth link between two computers need be established as in the simulation system described in section 3.1.1. The Bluetooth data are then channelled to a UWB routing driver interfacing to a real UWB radio link via a *WLP Convergence* layer, UWB Linux Driver and UWB hardware modules, instead of the gigabit Ethernet connection simulating a lower level UWB radio connection in section 3.1.1. Figure 11 provides the system diagram of this real BToUWB implementation.

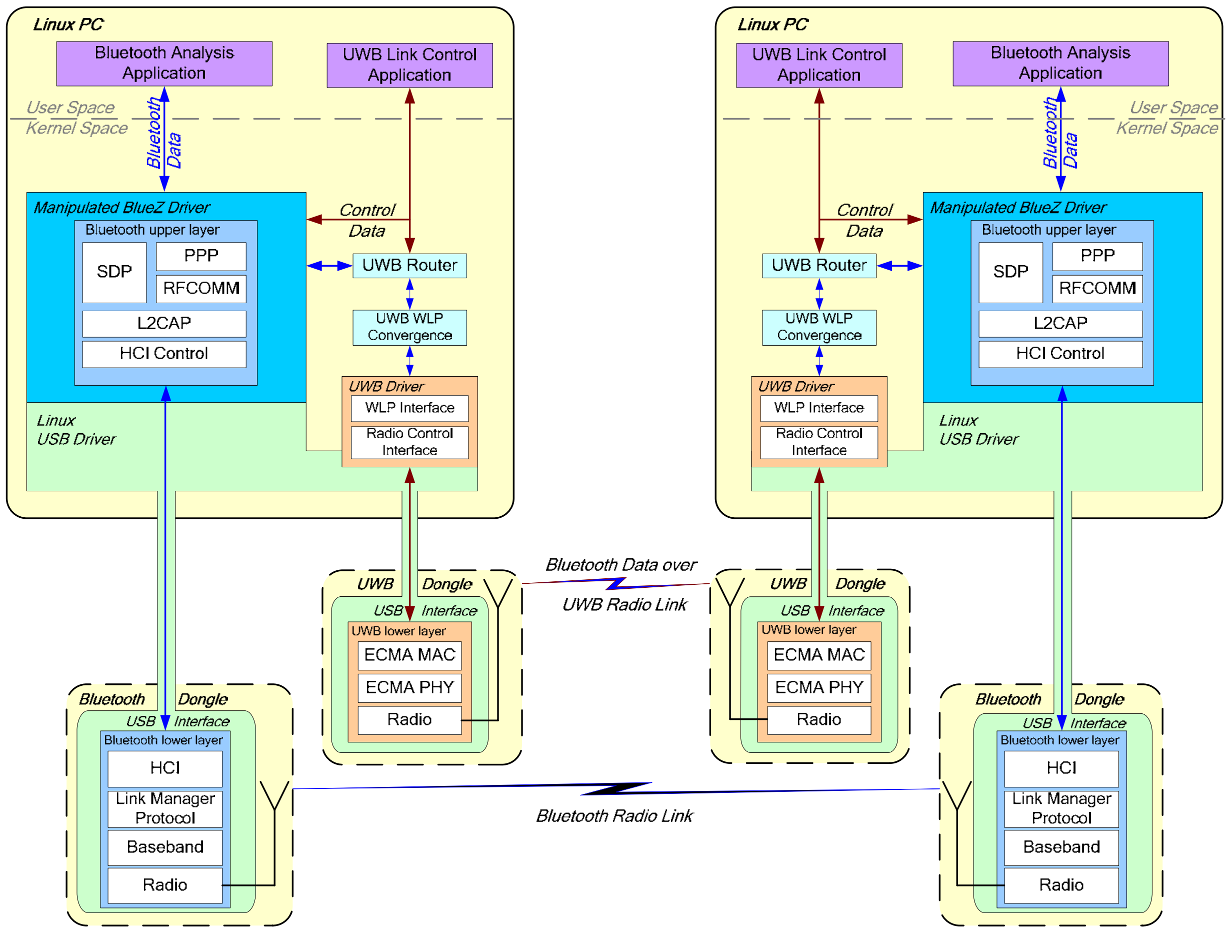

Figure 11: System diagram of Bluetooth over UWB implementation

As with the system in section 3.1.1, prior to UWB routing a fully compliant Bluetooth connection needs to be configured between two Bluetooth enabled computers under the open source Linux operating system [33] using the official Bluetooth stack for Linux called BlueZ [18]. This BlueZ protocol stack is described in more detail in section 3.2.1.

The same modification to the BlueZ HCI layer source files as in the simulation system were used to redirect L2CAP and SCO layer data to a routing layer, implemented via a kernel module named *UWB Router*. Section 3.2.5 provides a functional overview of this routing system. A *UWB Link Control* user level application was also developed to control the data routing performed by the routing subsystem. This application is also described further in section 3.2.5.

In turn, the *UWB router* interfaces to the official Linux UWB stack via a *UWB WLP Convergence* module. This module makes use of the UWB driver's WLP interface for accessing the UWB subsystem.

To generate valid Bluetooth data for massaging the links, a Bluetooth analysis program has been developed according to proposed Bluetooth user level methods [30]. Section 3.2.2 provides details on how to develop user level Bluetooth programs, followed by a detailed description of this analysis program.

Prior to routing, all drivers need to be loaded and both the Bluetooth and UWB links set up by using the utilities listed in Addendums A and C.

3.2 COMPONENT DESCRIPTION

This section provides a candid description of each of the components that make up the BToUWB system depicted in section 3.1. Each component is a significant subsystem on its own, with those components developed from scratch requiring a substantial amount of effort. For budgetary and accessibility reasons an open source Bluetooth stack were used for this research under the open source Linux operating system.

3.2.1 BlueZ Protocol Stack

The BlueZ protocol stack is an implementation of the Bluetooth wireless standard specifications under Linux. The code is licensed under the GNU General Public License (GPL). As of 2006, the BlueZ stack supports all core Bluetooth protocols and layers. BlueZ currently consists of the following separate modules, also shown in Figure 12:

- Bluetooth kernel subsystem core
- L2CAP and SCO audio kernel layers
- RFCOMM, BNEP, CMTP and HIDP kernel implementations
- HCI UART, USB, PCMCIA and virtual device drivers
- General Bluetooth and SDP libraries and daemons
- Configuration and testing utilities
- Protocol decoding and analysis tools

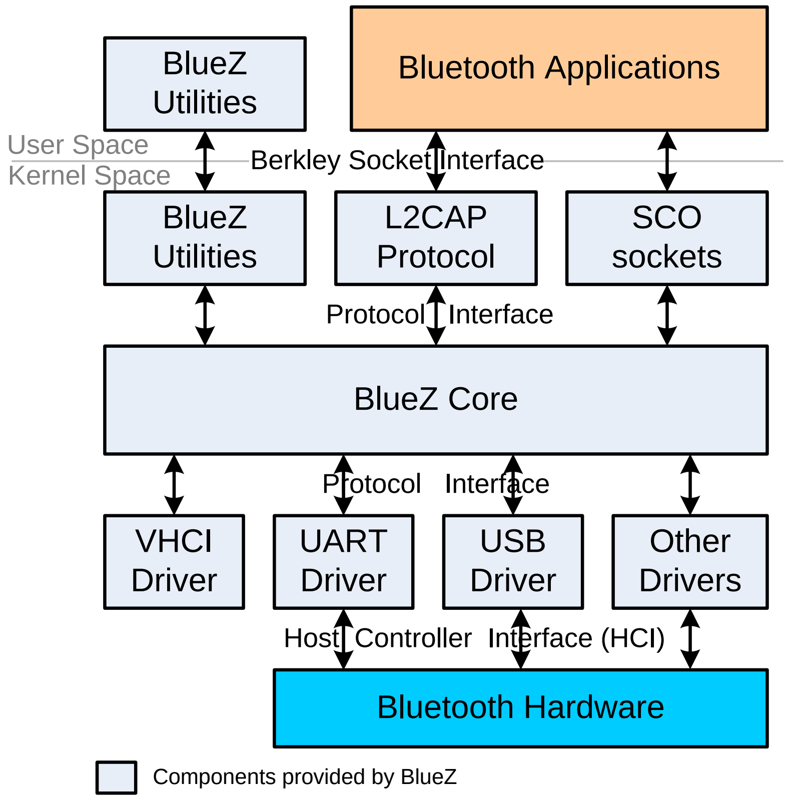

Figure 12: BlueZ Overview Diagram [21]

3.2.1.1 BlueZ setup

BlueZ is currently included in the Linux 2.4 and Linux 2.6 kernel series [18], and therefore already incorporated in most Linux distributions. Support for BlueZ can be found in these Linux systems on the market:

- Debian GNU/Linux
- Ubuntu Linux
- Fedora Core / Red Hat Linux
- OpenSuSE / SuSE Linux
- Mandrake Linux
- uCLinux (for embedded systems)

A number of online guides exist on how to setup and enable the BlueZ stack.

The BlueZ platform was chosen above investigated alternatives like PyBluez and Java, for its availableness, support and access to lower level Bluetooth functions [30]. BlueZ under Linux also provides extensive API's to fully exploit all local Bluetooth resources.

3.2.1.2 BlueZ tools and utilities

BlueZ provides a number of tools and utilities to configure and test Bluetooth protocol functionality and throughput. Table 18 lists the most important BlueZ utilities, with a complete description of the tools and utilities provided by the BlueZ stack elicited in Addendum A [21].

BlueZ utility	Description	Usage
`HCICONFIG`	an HCI device configuration utility	See Addendum A.1.2
`HCIATTACH`	an HCI UART driver utility that initialises a given serial port	See Addendum A.2.2
`HCITOOL`	used for generic writing to and monitoring of the HCI interface	See Addendum A.3.2
`HCIDUMP`	an HCI packet analyser	See Addendum A.4.2
`HCIEMU`	an HCI emulation daemon	See Addendum A.5.2
`L2PING`	a pinging utility for testing L2CAP connectivity	See Addendum A.6.2
`L2TEST`	a debug utility for testing L2CAP channel functionality and throughput	See Addendum A.7.2
`SCOTEST`	a debug utility for testing SCO channel functionality and throughput	See Addendum A.8.2
`RFCOMM`	a utility for establishing an RFCOMM connection over Bluetooth	See Addendum A.9.2

Table 18: BlueZ utility description

The abovementioned are all command-line based tools. The SuSE Linux distribution also offers a couple of Graphical User Interface (GUI) utilities for configuring, monitoring and using the Bluetooth subsystem provided by BlueZ.

GNOME Bluetooth is a graphical user interface to the Linux Bluetooth subsystem, licensed under the GNU GPL [19]. It consists of two packages:

- gnome-bluetooth: provides desktop level support for Bluetooth devices
- libbtctl: provides a GLib style library of application support for accessing Bluetooth under Linux.

The KDE counterpart to GNOME however offers more sophisticated utilities by the use of kdebluetooth [20]. These include the utilities listed in Table 19.

Application	Description
`kbluetooth`	Bluetooth Meta Server
`kbtsearch`	Bluetooth device/service search utility
`khciconfig`	KDE Bluetooth Monitor
`kioclient`	KIO command line client
`qobexclient`	Swiss army knife for obex testing / development
`kbtobexclient`	A KDE Bluetooth Framework Application
`kbtobexsrv`	KDE Obex Push Server for Bluetooth
`kbluepin`	A KDE KPart Application

Table 19: KDE Bluetooth Utilities

3.2.2 Bluetooth programming

In view of the available Bluetooth protocol described in section 3.2.1, custom Bluetooth programs for analysing and comparing the performance of various Bluetooth connections was developed in C under Linux.

Linux applications can access the Bluetooth subsystem via sockets. A number of different socket types are available, of which the higher level RFCOMM and lower level L2CAP are the most popular. From a programming point of view, RFCOMM sockets are closely related to TCP sockets, whereas L2CAP resembles UDP socket characteristics. L2CAP and RFCOMM are however logical higher level connections multiplexed on a single lower level radio connection. Adjusting the delivery semantics of one higher level connection will therefore affect all other active L2CAP and RFCOMM connections. In addition to RFCOMM and L2CAP sockets, the even lower level HCI socket is useful to gain a direct connection to the microcontroller on the local Bluetooth adapter. Furthermore SCO sockets provide a synchronous data channel, whilst BNEP (Bluetooth Networking Encapsulation Protocol) sockets encapsulate foreign network packet formats into a standard common format [22].

A full description of Bluetooth C source data structures and function calls used in Bluetooth programming is given in Addendum B.1 and B.2. Addendum B.3 provides an implementation where these concepts are put to use in a user level application that was developed partly for investigating Bluetooth programming techniques under Linux and partly for investigating Bluetooth data flow through the Linux subsystem. The command line driven application can setup server and client connections over Bluetooth's RFCOMM and L2CAP interfaces using sockets, and make use of the Service Discovery Protocol to probe the SDP entries of a remote device. The Bluetooth socket functions used for the developed research application were grouped together in a separate module called `btapi` for reuse in later developments. The source of this common Bluetooth socket interface module is listed in Addendum B.4.

Section 3.2.2.1 provides more information on the Bluetooth analysis program developed for the BToUWB system.

3.2.2.1 Bluetooth data generator and analysis program

Based on the abovementioned concepts and those listed in Addendums B.1 and B.2, a user level application called *btan* was developed to generate the different kinds of Bluetooth data to be sent over the various lower level routes. This program also provides a user level interface for changing the data types on the fly, and displaying system status updates and test results. Table 20 shows the command line options to be used with the developed *btan* analysis program in order to generate various Bluetooth data test scenarios:

Option	Usage
-c	Open in 'client' mode, followed by remote address: AA:BB:CC:DD:EE:FF
-t	Traffic Type: 'l'2cap, 's'co, 'r'fcomm
-h	Bluetooth channel for communication
-n	Number of packets to send
-d	Delay between packets
-z	Packet size
-i	Packet size incrementation
-j	Increment the packet size every 'j' packets
-f	Output file to log data in
-a	Audio file for SCO testing
-g	Average similar sized results
-v	Verbose

Table 20: *btan* developed Bluetooth analysis program options

The program access the Linux Bluetooth subsystem via socket connections for sending and receiving either asynchronous ACL data over the L2CAP layer, asynchronous serial data over the RFCOMM layer, or isochronous SCO streaming data over BlueZ’s SCO implementation.

In the complete sense of the term, isochronous data is transmitted at a fixed rate, and would therefore not directly benefit from a faster transmission medium. Unfortunately, due to limited access provided by the UWB hardware and firmware obtained, the true isochronous nature of the UWB TDMA access scheme could not be explored for SCO implementation. As alternative, the higher bandwidth UWB presents is applied to increase the Bluetooth isochronous link quality. This is achieved by tunnelling the SCO data over a more reliable UWB asynchronous data link, and buffering the data on the receiver side for timely insertion into the isochronous data stream. The link is more reliable due to the built in retransmission of corrupted data, whilst this is feasible thanks to the higher transmission speeds posed by UWB. For a 64 kbps Bluetooth SCO channel, a 480 Mbps UWB channel ought to keep the proposed buffer on the receiver side filled with reliable samples.

To implement this system, data to be transferred over the Bluetooth SCO connection were duplicated and transmitted to the remote device over the existing UWB link. These samples are then buffered on the remote device, to be inserted back into the existing SCO data stream at the original synchronous rate, replacing the Bluetooth SCO samples with inferior quality.

The different analysis performed on the Bluetooth data tranceived by this application are presented in section 3.3.

The source code for this developed application is provided in Addendum B.5.

3.2.3 Linux UWB Stack

The "Linux UWB + Wireless USB + WiNET" online development group provides a Linux based Ultra Wideband stack, as well as drivers for Wireless USB Host Controllers, UWB Radio Controllers and associated hardware [23]. At the time of this writing, driver development were still in progress and only supported a limited expensive range of UWB development kit hardware, including Alereon's AL4500 Development Kit and Intel's Wireless UWB Link 1480 (UWB radio, WiNET IP over UWB, Wireless USB Host Constroller) [24]. The developed Linux driver conforms to the Ultra Wideband standards set by the WiMedia Alliance [25] (as discussed in section 2.2) and also the ECMA-368 [26] and ECMA-369 [27] UWB PHY and MAC Standards (as discussed in sections 2.2.1.1 and 2.2.1.2).

3.2.3.1 Requirements

Building the Linux UWB stack requires a Linux capable PC with the following:

- A GCC v3.x or later compiler suite
- A Linux Kernel 2.6.18-rc6 or later source tree
- The Mercurial source control system [32] for patching the Kernel
- The Linux UWB driver [23]. The UWB Linux driver is distributed with a set of patches that have to be applied to the Linux kernel in order for the UWB USB support to work.

Release notes available online [35] contains all the required information to obtain, compile and install the driver. Addendum C gives an overview of the UWB utilities and commands used to implement the UWB driver via commands given to file system entries created for each UWB hardware device enumerated.

3.2.3.2 Device Discovery

By using the *scan* and *beacon* commands created by the driver, two remote UWB devices can advertise and scan beacons on an agreed upon channel. If there were other devices in the same radio channel and beacon group, the dongle's radio control interface will send beacon notifications on its notification/event endpoint (NEEP). The HWA-RC driver listens for notifications, parses it and enrols it in the UWB daemon queue.

When the UWB daemon wakes up and scans the event list, it finds a beacon and adds it to the beacon cache. After a number of beacons are received from the same device, it is considered to be 'onair' and a new uwb device is created for it. Similarly, when no beacons are received for some time, the device is considered gone and purged from the system.

3.2.3.3 Data Transmission

Data is sent and received through remote pipes. A remote pipe is aimed at an endpoint in a WUSB device. Each HC has a number of remote pipes and buffers that can be assigned to them. Data buffers are segmented out before sending. When the transfer is executed, a notification that says data is ready is received in the DTI endpoint, providing a descriptor giving the status of the transfer, its identification and the segment number. If the transfer was a data read, a USB Request Block (URB) is issued to read into the destination buffer the data produced by the remote endpoint.

For OUT transfers, an array of segments exists. When submitting, URBs for segment request 1, buffer 1, segment 2, buffer 2 etc. are used. When all the segments are complete, a callback function is called to finalise the transfer. For IN transfers, URBs are only issued for the segments that need to be read.

3.2.3.4 UWB Driver Implementation

For this project the openSuSE 10.2 Linux distribution [33] were used with GCC version 4.1.2 and kernel version 2.6.23.17 patched against the Linux UWB driver version 1.8.9. Recompiling a Linux kernel is out of the scope of this document, but numerous online guides exists for each distribution flavour. A detailed guide for patching and compiling the Linux kernel under SuSE 10 is available [34].

Instructions on building the Linux UWB driver against the patched kernel source tree and installing the modules can be found in the Linux UWB v1.8.9 release notes [35]. In order for the IO Gear UWB dongles used in this project to function under the Linux UWB driver control, some modifications to the Linux UWB source are required prior to building it, forcing it to be compliant to WHCI 0.96 and above. Upon connecting the IO Gear dongles, it is important to obtain the following kernel debug messages:

```
kernel: usb 3-1: new high speed USB device using ehci_hcd and address 2
kernel: usb 3-1: configuration #1 chosen from 1 choice
```

The HWA-RC device also fails to bind automatically to the IO Gear dongles and requires re-loading of the drivers. When successfully binding to the hardware, the kernel will display the assigned MAC and device addresses, similar to this:

```
uwb-rc uwb0: new uwb radio controller (mac 00:17:c4:0a:59:12 dev 17:d3) on
usb 3-1:1.0
```

And will create devices visible in `/sys/class/uwb_rc/` and `/sys/class/uwb/`.

UWB utilities described in Addendum C should then be used to initiate beaconing by one device whilst scanning by the other. By following this process, the two devices can then join the same beacon group, as described in section 2.2.1.2.

3.2.4 UWB Simulation subsystem

Since the focus of the project shifted to the physical implementation of a UWB subsystem, only a brief overview of this alternate simulation subsystem will be provided here.

A layout of the developed simulation system and how its subsystems interconnect with each other is given in Figure 13.

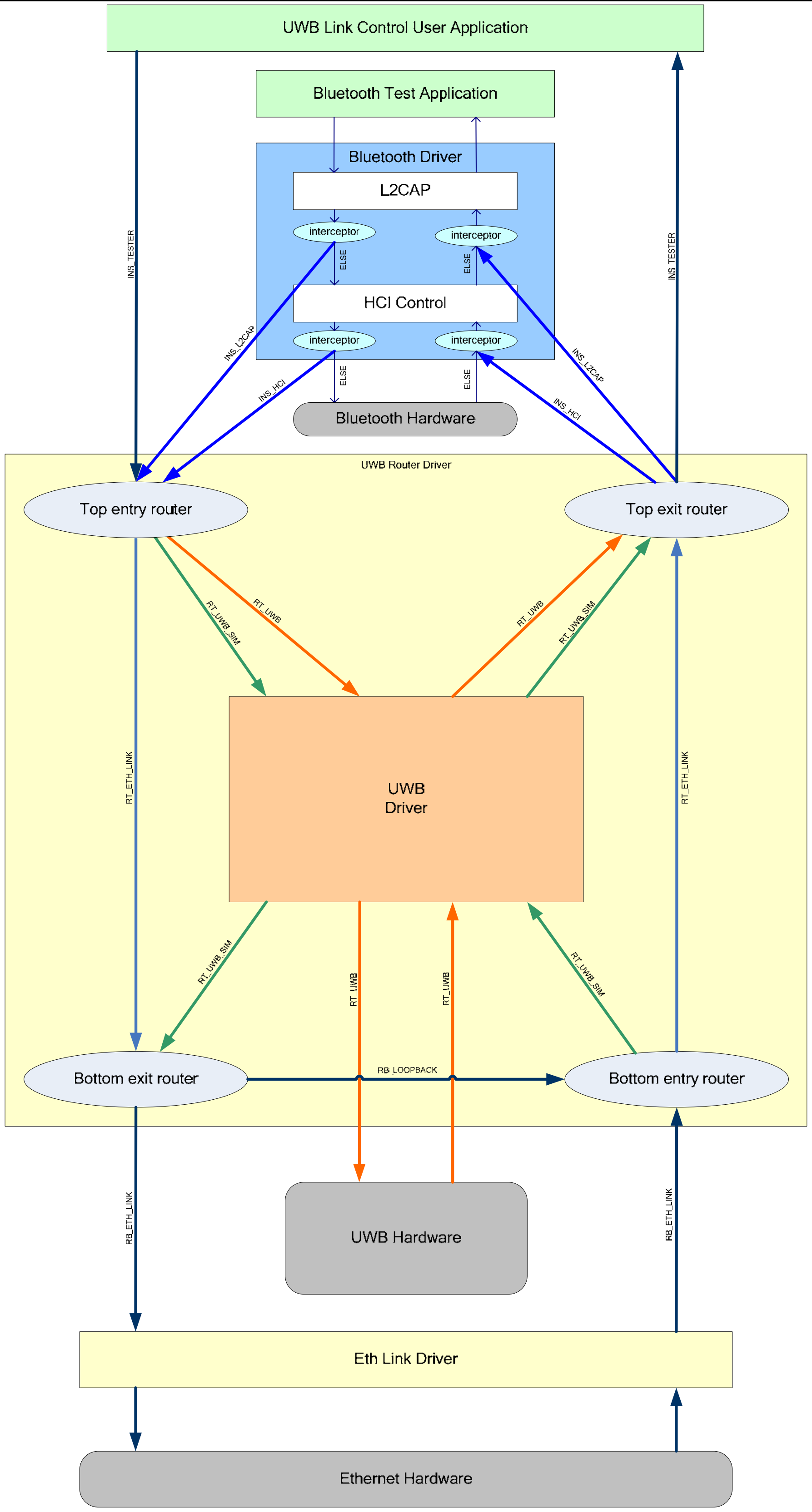

Figure 13: UWB Simulation data transfer and system control layout

A modified radio controller was added to the Linux UWB driver system (see section 3.2.3.4) by spoofing a `uwb_dev` and `uwb_rc` structure to be registered with the driver. The UWB simulator generates the minimum responses to fake a UWB Radio Controller to the UWB driver, whilst dropping packets at random (dependant on a predefined rate) to simulate interference. Medium access contention and transceiver distances were not simulated. A thread waking up periodically to handle transfers simulates a DRP channel.

The *WLP Convergence* layer in turn makes direct calls to the Linux UWB driver to send and receive data through the modified UWB stack. The simulated Radio Controller (RC) encapsulates and sends the UWB RC’s frames over the Ethernet connection link from computer A to B, where it is introduced back into B’s UWB driver via an RC notification, and propagated back up to the *WLP Convergence* layer through call-back functions.

3.2.5 UWB Routing

Data transfer between and control of the system components described in section 3.2 are handled from a single integrated kernel module, called *UWB Router*, sitting at the heart of the system. Figure 14 depicts this dataflow between the system components dependant on control signals setup by the *UWB Router* driver. The complete C source of the developed module is listed in Addendum D.1 as detailed description and for reproducing the system.

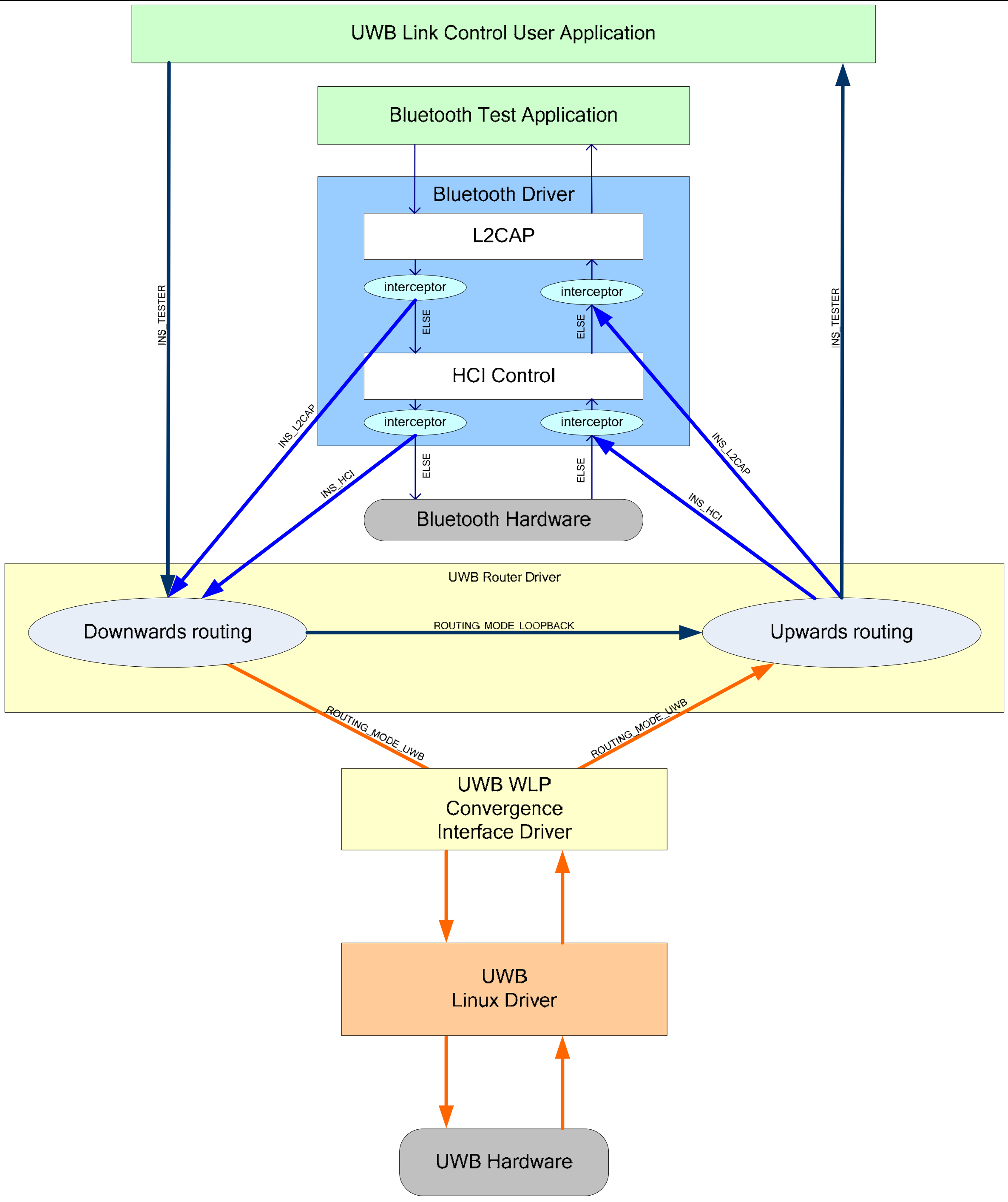

Figure 14: UWB Router Module Functional Layout

Table 21 lists the options provided by the *UWB Router* for obtaining source Bluetooth data packets to channel over the BToUWB link:

Code declaration	Control key	Description
`INS_MODE_L2CAP`	`c`	L2CAP Insertion Point
`INS_MODE_HCI`	`h`	HCI Insertion Point
`INS_MODE_TESTER`	`t`	UWB Link Control Test Data

Table 21: UWB Router insertion channel options

The *UWB Router* also provides the options listed in Table 22 for choosing over which available lower level channels to send these packets:

Code declaration	Control key	Description
`ROUTING_MODE_CLOSED`	`n`	No Routing
`ROUTING_MODE_UWB`	`u`	UWB WLP Routing
`ROUTING_MODE_ETH_LINK`	`p`	Ethernet Simulation Routing
`ROUTING_MODE_LOOPBACK`	`l`	Local Loopback Routing

Table 22: UWB Router lower layer data routing options

The actual packet routing is performed by the `uwb_router_downwards_routing()`and `uwb_router_upwards_routing()` functions. Packets are routed downwards to lower layers based upon the channel selection as listed in Table 22. A 4 byte header is added to the packet for signifying its channel of origin to the receiver side for upwards routing. The receiver extracts this header from the packet and routes it to upper layers based upon the embedded header information as listed in Table 22.

To enable the *UWB Router* kernel module to transmit data to and receive data from other Linux kernel modules (like the Bluetooth stack and *WLP Convergence* module), source code were added into these modules to register a callback function to the *UWB Router* module. Upon a data event, these callback functions then executes the correct data handler inside the *UWB Router* module.

Following is a brief description on how the *UWB Router* interfaces to the different system components as well as the developed user application that controls the module's behaviour.

3.2.5.1 Bluetooth Connection

At the Bluetooth layer HCI commands sent to the HCI sublayer by upper layer L2CAP and SCO implementations were hijacked by modifying the open source BlueZ files. These packets were then sent to the *UWB Router* implementation for transmission over a UWB hardware subsytem. Similarly lower level HCI events are spoofed to the L2CAP and SCO layers by the *UWB Router* implementation upon receiving packets from the UWB hardware subsystem.

The modified L2CAP and SCO Bluetooth layers pass the hijacked data buffers on to the *UWB Router* layer after adding a header containing specific data to spoof an HCI connection on the receiver's side. Specifically calls made to the `hci_send_acl()` function in the `l2cap.c` file and `hci_send_sco()` function in the `sco.c` file were targeted. To propagate data back into the Bluetooth system, a new HCI connection is set up upon a receiving event by using and removing the added header data followed by calls to either the `l2cap_recv_acldata()` or `sco_recv_scodata()` functions.

Addendum D.4.1 shows the detailed modifications performed to the BlueZ L2CAP layer, whilst addendum D.4.2 does the same for the BlueZ SCO module.

3.2.5.2 UWB WLP Connection

The *UWB Router* layer in turn makes use of the WLP interface provided by the official Linux UWB driver to send and receive data through the UWB stack. A separate module named *WLP Convergence*, described in section 3.2.6, was developed to handle interfacing to the WLP UWB layer.

The `uwb_wlp_conv_tx_packet_top_entry()` function is used as entry point to the *WLP Convergence* module for passing data packets to it. Prior to receiving packets from the *WLP Convergence* module, a receiving function to be called on data reception needs to be registered with the module by use of the `uwb_wlp_conv_register_receiver_function()`. The `uwb_wlp_conv_rx_packet_top_exit()` function then acts as the exit point for data packets from the *WLP Convergence* module to be sent to the registered receiver function.

Due to the implementation of *WLP Convergence* functions in the *UWB Router* module, the *WLP Convergence* module needs to be loaded into the system prior to loading the *UWB Router* module.

3.2.5.3 UWB Link Control User Application

A user level command line based control application called *UWB Link Control* has been developed to provide a management interface to the different routing options provided by the *UWB Router* kernel module. The application opens a control interface to the *UWB Router* module described in section 3.2.5, allowing the user full control over the BToUWB data path by changing the Channel Insertion Mode (Table 21) and the lower layer Routing Mode (Table 22).

The application also provides the means to insert test data packets into the BToUWB data path for debugging purposes of the data channel and operations performed on the data packets. Packets of various sizes containing random data can be generated and send to the *UWB Router* for further downwards routing.

Prior to using this application the *UWB Router* module needs to be loaded on the system. For a comprehensive description of the application's functioning see its source code listed in Addendum D.3.

3.2.6 WLP Convergence

For setting up a transmitting and receiving connection to the UWB Linux Driver's WLP interface, and handling the sending and asynchronous arrival of packets over these connections, a kernel module named *WLP Convergence* was developed. This driver acts as interface between upper layer router (and Bluetooth) modules and lower layer UWB device drivers.

After setting up a WLP supported Ultra Wideband Link, the module can be loaded with the target device's WLP IP address as module parameter, e.g.;

```
insmod uwb_wlp_conv.ko uwb_wlp_ip_address="192.168.2.1"
```

Upon loading the driver will:

a) Open a server socket over WLP to the local UWB hardware device.
b) Create a thread to attempt opening a client socket over WLP to the IP address related target UWB device to be used for transmission.

c) Create a thread for the server receiving socket which will listen for incoming connections from remote UWB hardware to be used for receiving data from the remote UWB based device.

Upon establishing a connection to the remote WLP supported UWB based device, the `uwb_wlp_conv_tx_packet_top_entry()` function may be used to transmit packets over this link, and the `uwb_wlp_conv_register_receiver_function()` function may be used to register a call-back function to handle packets received over this link. The `uwb_wlp_conv_rx_packet_top_exit()` function then passes a received data packet to this registered receiver function upon receipt.

Addendum D.2 provides the detailed source files of this module for further explanation and reproduction.

3.3 ANALYSIS

The performance of the developed BToUWB system were analysed in terms of asynchronous data throughput, described further in section 3.3.1, and isochronous data reliability, elicited in section 3.3.2.

3.3.1 Bluetooth Asynchronous Data Analysis

The BlueZ utility L2PING sends data of programmable size back and forth over a Bluetooth link and reports the time that the transmission took. This tool can therefore be used to analyse Bluetooth data throughput of different sized packets.

In order to gain more control over this process (e.g. real-time varying of packet size and on-the-fly processing of data), a user level application performing a similar task was developed for remotely bouncing back data packets for analysis. As described in section 3.2.2.1, the *btan* Linux application will generate data for asynchronous data analysis over the Bluetooth and BToUWB systems by performing the steps listed in Table 23:

Bluetooth Client Connection	Bluetooth Server Connection
Check if the provided parameters / options is valid	
Setup the particular Bluetooth L2CAP or RFCOMM connection to the remote device	
Generate test data buffers of various lengths en transmit them to the server side of the connection	
	Receive and loopback the test data buffer to the client side
Receive the loopbacked test data buffer from the server, compare it to the transmitted buffer and calculate the elapsed time since sending the data	
Calculate the data throughput, and log it together with other Bluetooth radio parameters (e.g. radio signal strength and link quality) for later analysis	

Table 23: Bluetooth asynchronous data analysis performed by the *btan* application

Addendum B.5 provides the source code as a detailed explanation of how the application performs these tasks.

Prior to applying the analysis tool mentioned above, all the necessary BToUWB system components depicted in section 3.2 needs to be loaded and set up.

3.3.1.1 L2CAP ACL Test Data Acquisition

To investigate the affect of distance on the two wireless protocols, the same sets of data were acquired at various transmission distances, which include 0, 1, 2 and 3 meters. At each distance, 4 sets of data were logged:

i. A fair amount of small sized packets:

```
server_pc>./btan
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 5 -n 100 -h 5555
```

ii. A fair amount of medium sized packets:

```
server_pc>./btan
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 50 -n 100 -h 5557
```

iii. A fair amount of large sized packets:

```
server_pc>./btan
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 500 -n 100 -h 5559 -d 1
```

The extra -d option is for adding a delay for clean handling of the larger data packets through the system, especially the last one.

iv. A large amount of variable sized packets:

```
server_pc>./btan
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 4 -n 3250 -j 5 -i 1
           -h 5555 -d 1 -g
```

With the variable sized data set, transmission were started with a packet size of 4 bytes, and incremented every 5 packets with a single byte, until it reached 652 bytes.

3.3.1.2 RFCOMM Test Data Acquisition

Due to the fact that the Bluetooth RFCOMM layer implements the lower layer L2CAP to channel data, the same data analysis as depicted in section 3.3.1.1 above may be used. Similar results to L2CAP will also be expected, for the data flow through the Bluetooth (and UWB) subsystem is identical, with subtle timing differences in the connection setup.

To investigate the affect of distance on the two wireless protocols, the same sets of data were acquired at various transmission distances, which include 0, 1, 2 and 3 meters. At each distance, 4 sets of data were logged:

i. A fair amount of small sized packets:

```
server_pc>./btan –t r
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 5 -n 100 -t r -h 15
```

ii. A fair amount of medium sized packets:

```
server_pc>./btan –t r
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 50 -n 100 -t r -h 16
```

iii. A fair amount of large sized packets:

```
server_pc>./btan –t r
client_pc>./btan -c 00:0A:94:00:1A:EC -f log.txt -z 500 -n 100 -t r -h 17 -d 1
```

The extra -d option is for adding a delay for clean handling of the larger data packets through the system, especially the last one.

iv. A large amount of variable sized packets:

```
server_pc> ./btan -t r
client_pc> ./btan -c 00:0A:94:00:1A:EC -f log.txt -z 4 -n 3250 -j 5 -i 1
           -t r –h 15 -d 1 -g
```

With the variable sized data set, transmission were started with a packet size of 4 bytes, and incremented every 5 packets with a single byte, until it reached 652 bytes.

3.3.2 Bluetooth Isochronous Data Analysis

To analyse the isochronous link quality of a Bluetooth SCO connection, one cannot just compare sent data samples with received data samples, as was the case with ACL data. Bluetooth devices have embedded algorithms for transmitting voice over SCO, encoding sound received from the host into a specific "air" format, such as PCM u-law, PCM A-law, PCM linear, etc. CVSD is a lossy format which encodes differences from a predicted signal, and consequently it is impossible to detect carbon copy data at the receiver end. To analyse the SCO link quality, a comparison needs to be made between the sent and received data samples, correlating the samples and calculating the signal loss.

3.3.2.1 SCO Test Data Acquisition

The SCO channel supports 64kbps voice data. The data should be signed PCM values, which forces it to be 16-bit samples on each channel. The SoX cross-platform command-line audio transformation utility [37] were used to convert wave audio samples to the required Bluetooth PCM format for over-the-air transmission, and back to wave format for analysis.

To investigate the affect of distance on the two wireless protocols, the same sets of data were acquired at various transmission distances, which include 0, 1, 2, 3 and 4 meters. At each distance, 2 sets of data were logged:

i. Small low quality audio samples

```
client_pc> sox -t wav -r 128000 -s -w -c 2 small_in.wav -t raw small_in.raw
server_pc>./btan -t s -a small_out.raw
client_pc> ./btan -c 00:0A:94:00:1A:EC -t s -a small_in.raw
server_pc>sox -t raw -r 8000 -s -w -c 2 small_out.raw -t wav small_out.wav
```

ii. Large high quality audio samples

```
client_pc>sox -t wav -r 705000 -s -w -c 2 large_in.wav -t raw large_in.raw
server_pc> ./btan -t s -a large_out.raw
client_pc> ./btan -c 00:0A:94:00:1A:EC -t s -a large_in.raw
server_pc> sox -t raw -r 44100 -s -w -c 2 large_out.raw -t wav large_out.wav
```

The received audio signal is then compared to the transmitted audio signal in order to determine the signal quality loss during transmission. First of all, the time that the received signal is offset compared to the transmitted signal needs to be eradicated by correlating the two signals. Figure 15 shows the uncorrelated received (RX) audio signal compared to the transmitted (TX) audio signal.

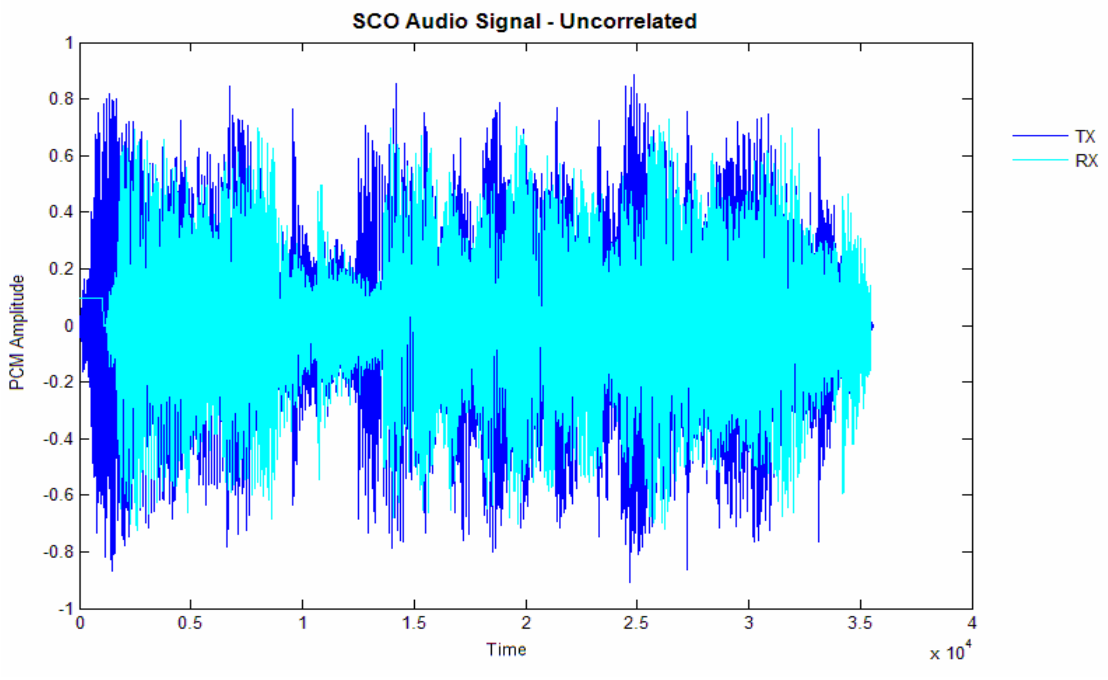

Figure 15: Received signal offset in time compared to the transmitted signal

Figure 16 shows the received (RX) audio signal after its correlated to the transmitted (TX) audio signal.

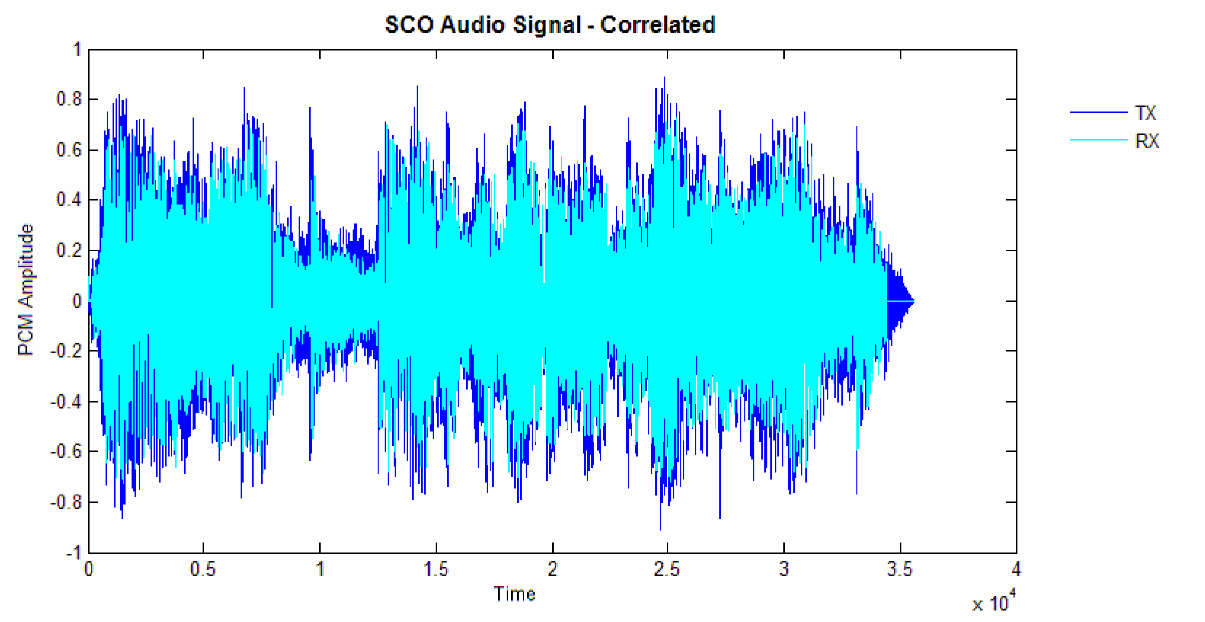

Figure 16: Received audio signal correlated to the transmitted audio signal

After the signals are correlated, the Peak Signal-to-Noise Ration (PSNR) is calculated to evaluate the quality of the received audio signal. This is achieved by first calculating the Mean Square Error (MSE) between the two signals:

$$MSE_{SCO} = \frac{1}{m}\sum_{i=0}^{m-1}\left|Tx_{SCO}(i) - Rx_{SCO}(i)\right|^2 \tag{3}$$

Where Tx(i) is the transmitted audio samples, Rx(i) is the received audio samples, and m is the number of audio samples transmitted. The PSNR is then calculated by the following equation:

$$PSNR_{SCO} = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE_{SCO}}\right) \tag{4}$$

CHAPTER 4: RESULTS

4.1 BLUETOOTH OVER UWB SIMULATION

Although the simulation system described in section 3.1.1 was replaced by a working hardware system, some results obtained are listed here to prove its functionality. Figure 17 shows a comparison between the measured transfer speeds of asynchronous L2CAP ACL data over a traditional Bluetooth only link using DH3 packets against that obtained from the simulated BToUWB system link. The BlueZ L2PING command were used to measure the turnaround time for different sized data packets between the two systems in Figure 9.

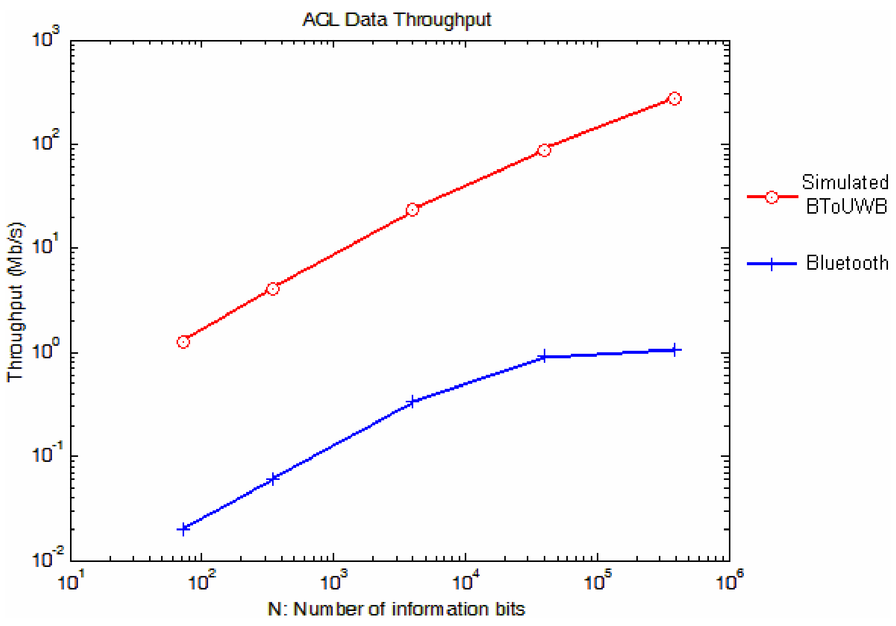

Figure 17: ACL data throughput for BT and simulated BToUWB

These results show that replacing the 1 Mbps radio bottleneck of Bluetooth with a simulated BToUWB subsystem dependent on an Ethernet link capable of 100 Mbps throughput, can provide a 100 fold improvement of the Bluetooth hosted application's bandwidth. Due to the larger packet capabilities of Ethernet, the flattening witnessed in the Bluetooth curve for packets exceeding 100 kilobytes (probably due to fragmentation overhead) is not prevalent in the simulated BToUWB results obtained.

4.2 BLUETOOTH OVER UWB SYTEM IMPLEMENTATION

This section lists the results obtained from analysing ACL and SCO data sent via the Bluetooth over UWB system implementation as described in section 3.1.2.

4.2.1 Bluetooth Asynchronous Data Analysis

4.2.1.1 L2CAP ACL Data : Bandwidth vs Packet Size

The results depicted in this section shows a comparison between the measured data throughput of asynchronous L2CAP ACL data over a traditional Bluetooth only link using DH3 packets against that obtained from the implemented BToUWB system described in section 3.1.2 for various transmission distances. The Bluetooth data generator program *btan* described in section 3.2.2.1 were used to send and loopback data over the two wireless links, using the analysis techniques described in 3.3.1.1 to obtain the depicted results. Figure 18, Figure 19, Figure 20 and Figure 21 illustrate the ACL data throughput (bandwidth) as a function of the ACL packet size over a 0, 1, 2 and 3 meter transmission distance respectively.

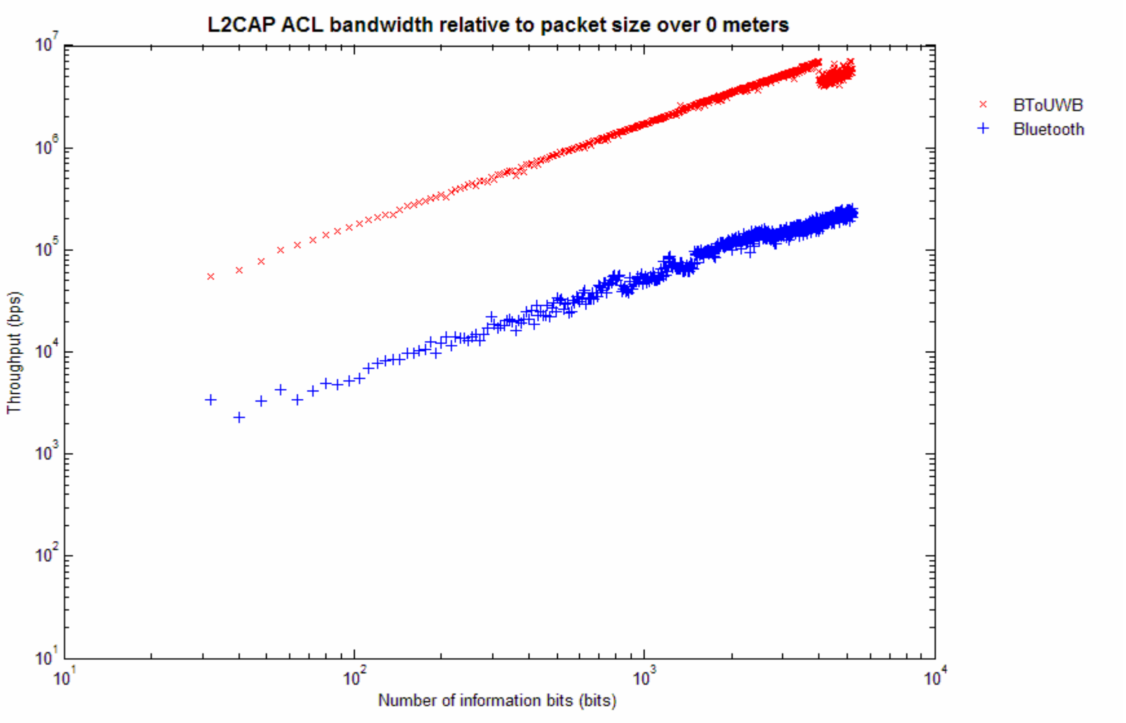

Figure 18: ACL data throughput over 0 m transmission distance

Figure 18 shows how the higher bandwidth offered by the UWB subsystem is reflected in the BToUWB 50 fold throughput increase over the traditional Bluetooth bandwidth for various packet sizes at 0 meter transmission distance. A positive sloping graph depicts how the bandwidth is more optimally utilised with increasing packet sizes: a larger proportion of the each packet contains actual payload data versus overhead included. The discontinuity witnessed in the BToUWB results at around 3500 bits is explained in section 4.2.1.5.

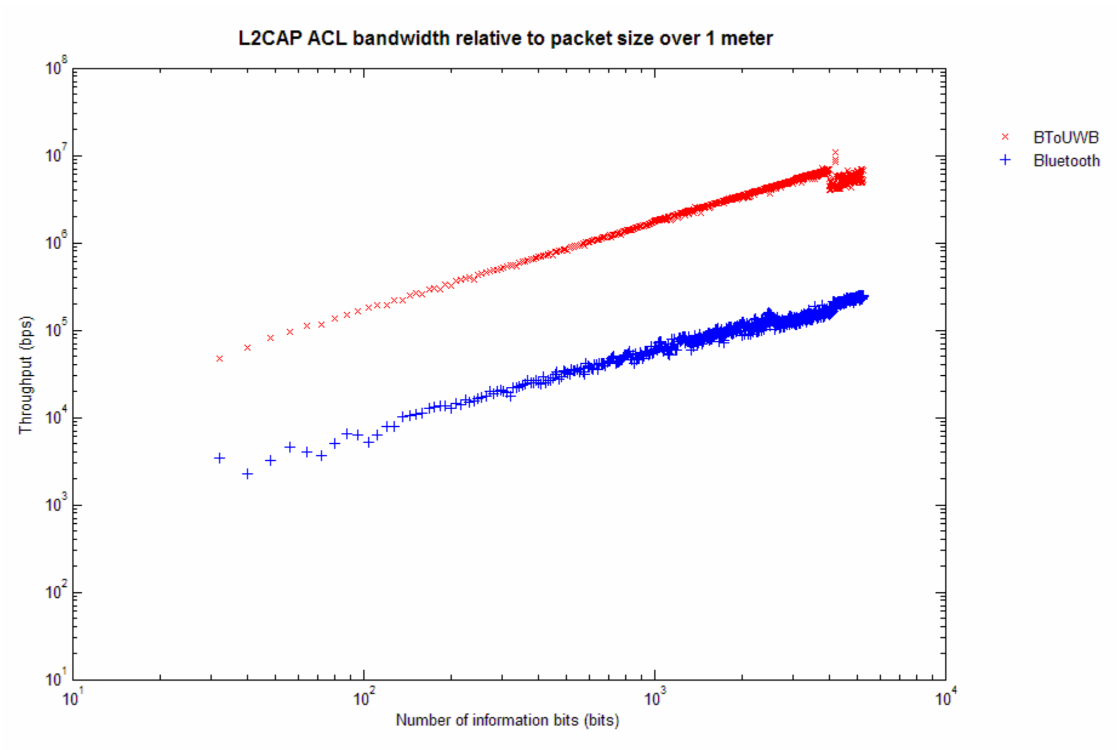

Figure 19: ACL data throughput over 1 m transmission distance

Figure 19 depicts a similar 50 fold increase in bandwidth observed for the BToUWB system over 1 meter transmissions, attributed to the increased throughput provided to the total system by the underlying applied UWB hardware. Once again a discontinuity observed at 3500 bits can be ascribed to the overhead explored in section 4.2.1.5.

At 2 meters, showed in Figure 20, similar bandwidth improvements are observed for the BToUWB system. The detrimental effect that the increased transmission distance is starting to have on the Bluetooth link can also be observed.

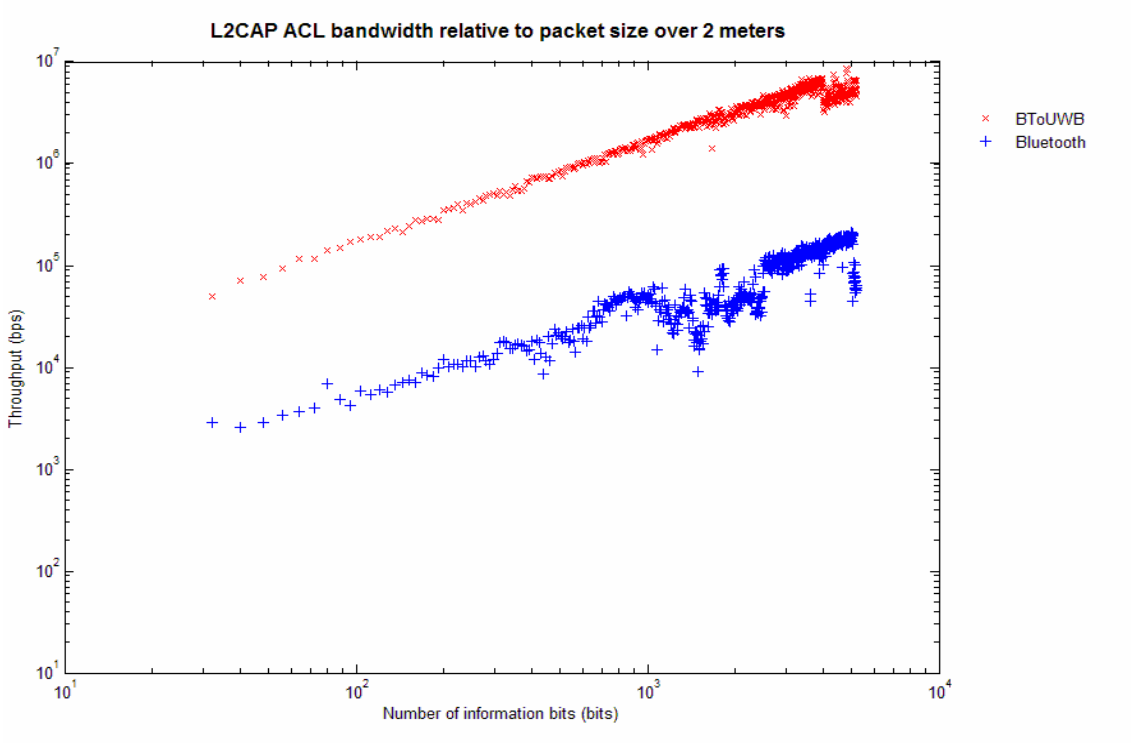

Figure 20: ACL data throughput over 2 m transmission distance

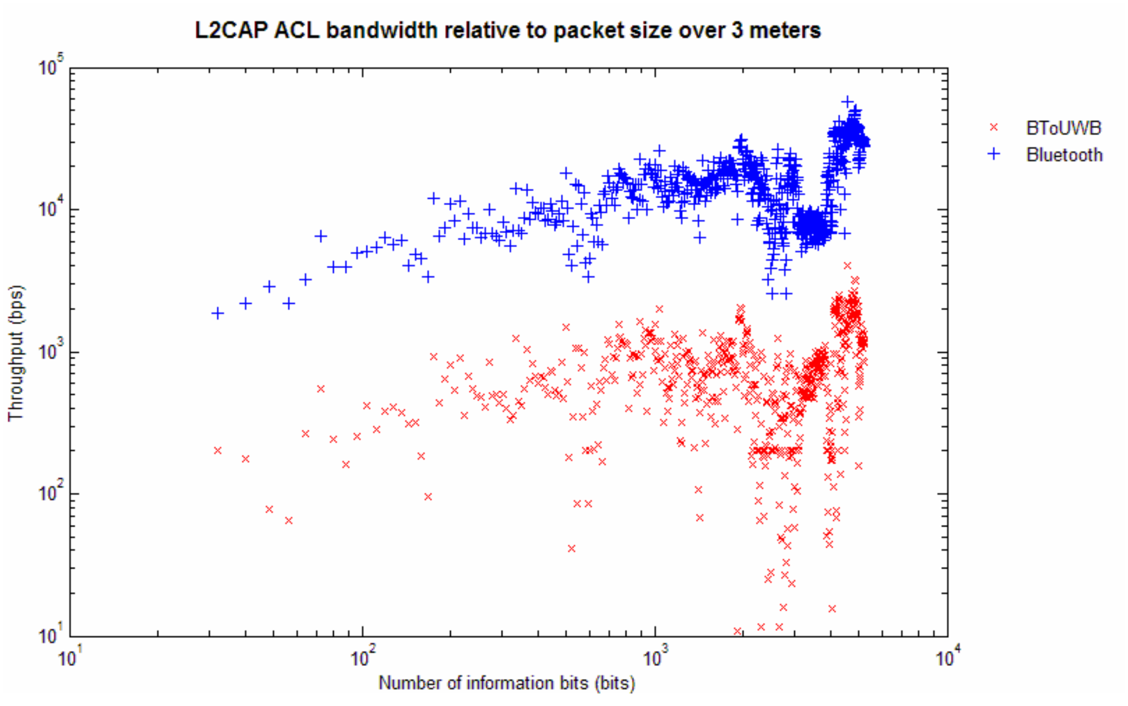

Figure 21: ACL data throughput over 3 m transmission distance

Once the transmission distance were increased to 3 meters, both the BToUWB and Bluetooth channels started to decay in quality, with very bad performance around 3300 bits, as shown in Figure 21. The reason for this loss of bandwidth is explored in section 4.2.1.6.

4.2.1.2 L2CAP ACL Data : Bandwidth vs Transmission Distance

Taking another perspective, Figure 22, Figure 23 and Figure 24 illustrate the ACL data throughput (bandwidth) as a function of the distance between the wireless transceivers for small, medium and large ACL packet sizes respectively.

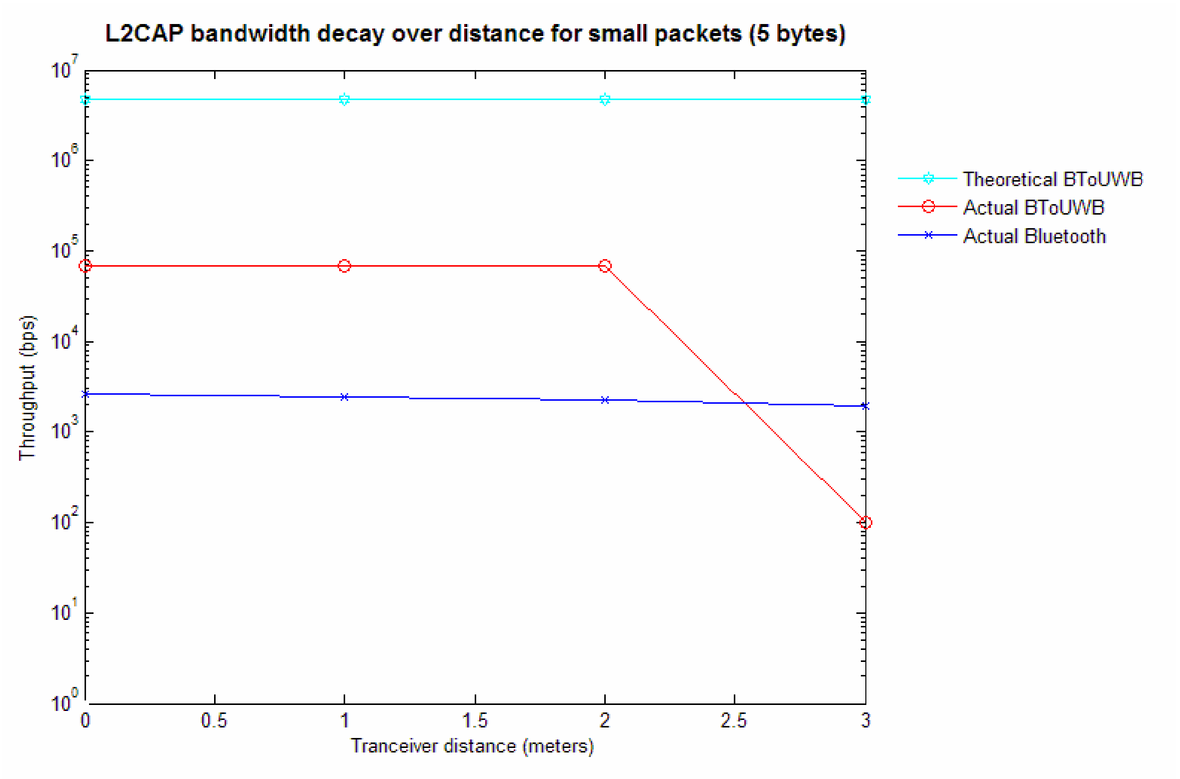

Figure 22: ACL data throughput vs. transmission distance for small packets

Figure 22 shows how only a slight decay in Bluetooth bandwidth (at this scale) can be observed over 0 to 3 meter transmission distance. A fairly constant throughput were obtained for 0 to 2 meter BToUWB transmissions, with a sharp decay in throughput above 2 meters, as also seen in Figure 21, where this underperformance are described in paragraph 4.2.1.6. The theoretical BToUWB value shown in Figure 22 were obtained from calculation depicted in section 4.2.1.7.

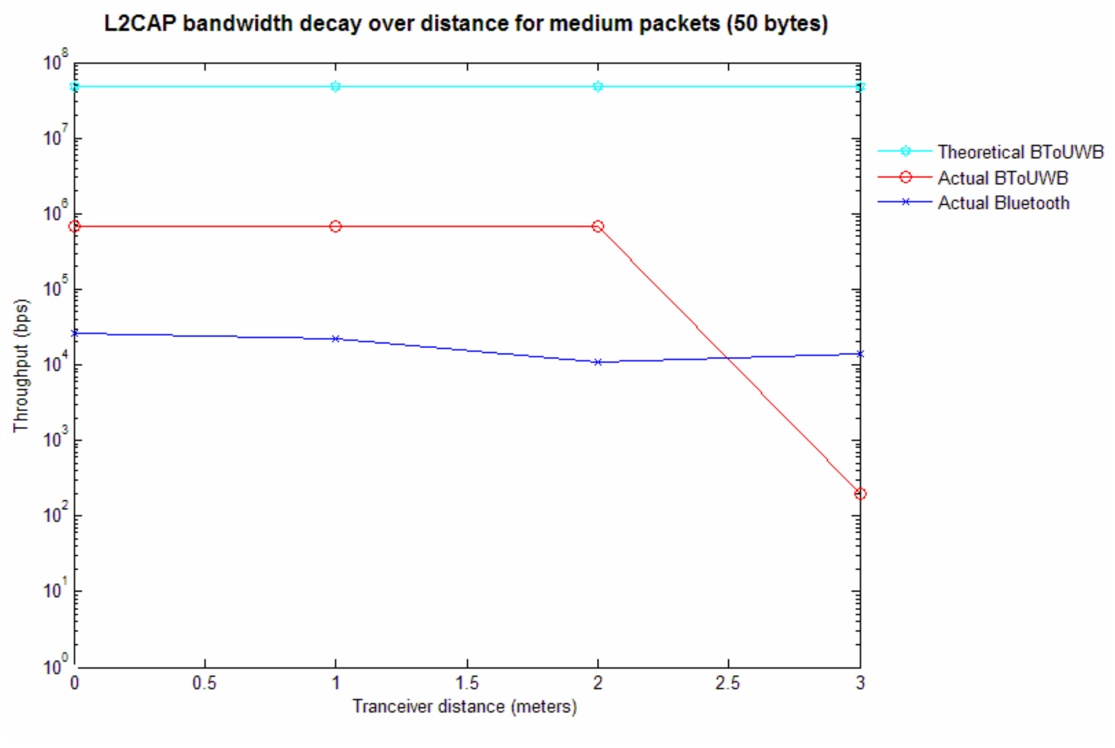

Figure 23: ACL data throughput vs. transmission distance for medium packets

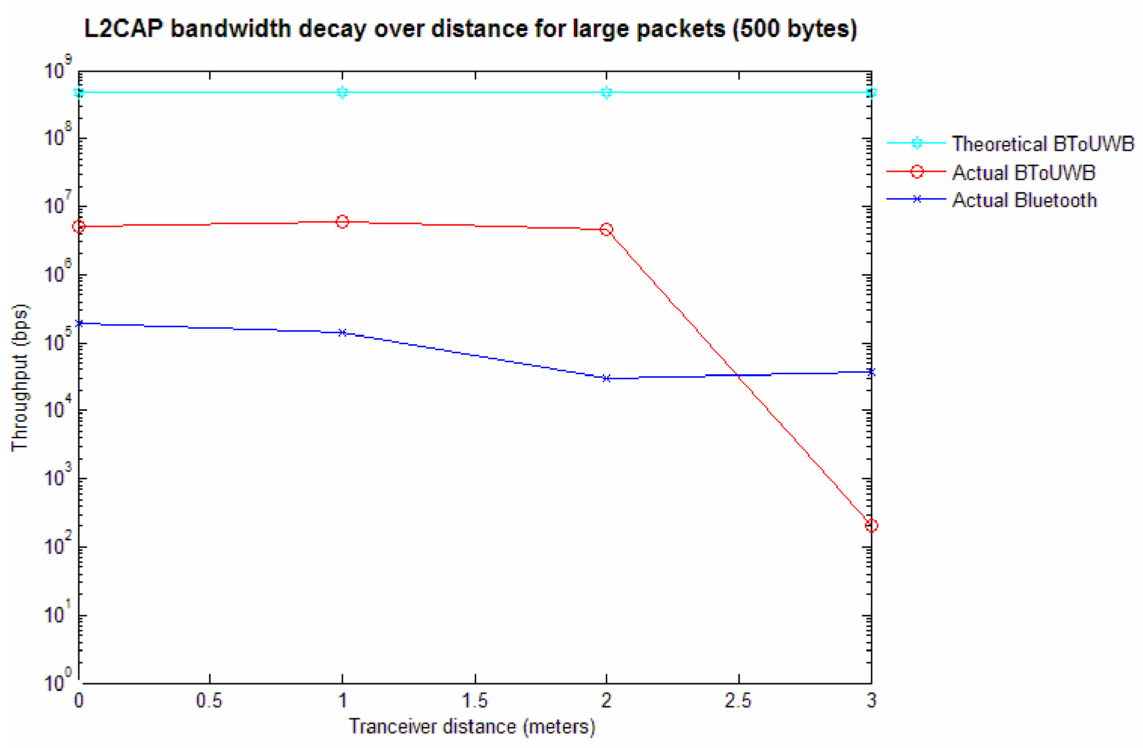

Figure 24: ACL data throughput vs. transmission distance for large packets

Similar to Figure 22, Figure 23 and Figure 24 shows a sharp corrosion of bandwidth as soon as the transmission distance increased above 2 meters. See section 4.2.1.6 for an explanation of this bandwidth loss, and section 4.2.1.7 for the rationalisation behind the two theoretical BToUWB bandwidth figures used for comparison in Figure 23 and Figure 24.

4.2.1.3 RFCOMM Data: Bandwidth vs Packet Size

The results depicted in this section shows a comparison between the measured data throughput of asynchronous RFCOMM data over a traditional Bluetooth only link using DH3 packets against that obtained from the implemented BToUWB system described in section 3.1.2 for various transmission distances. The Bluetooth data generator program *btan* described in section 3.2.2.1 were used to send and loopback data over the two wireless links, using the analysis techniques described in 3.3.1.2 to obtain the depicted results. Figure 25, Figure 26, Figure 27 and Figure 28 illustrate the RFCOMM data throughput (bandwidth) as a function of the RFCOMM packet size over a 0, 1, 2 and 3 meter transmission distance respectively.

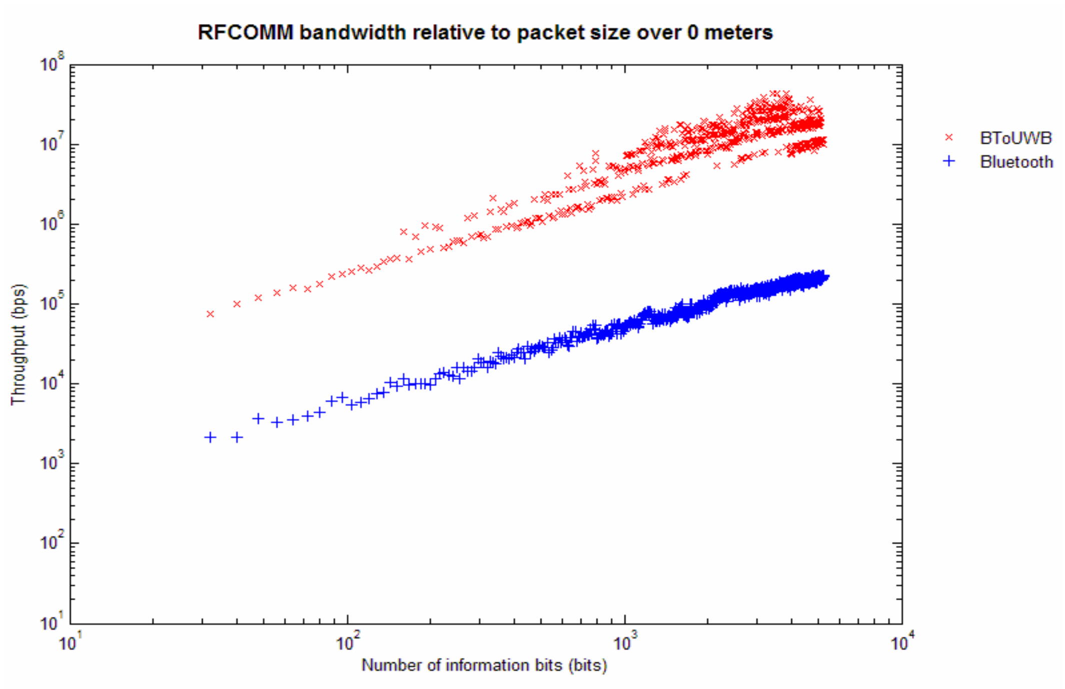

Figure 25: RFCOMM data throughput over 0 m transmission distance

As for the L2CAP results depicted in section 4.2.1.1, Figure 25 shows how the higher bandwidth offered by the UWB subsystem is reflected in the BToUWB 50 fold throughput increase over the traditional Bluetooth bandwidth for various packet sizes at 0 meter transmission distance. A positive sloping graph depicts how the bandwidth is more optimally utilised with increasing packet sizes: a larger proportion of the each packet contains actual payload data versus overhead included. The discontinuity at 3500 bits obtained in the L2CAP results is not as prevalent in this RFCOMM data acquire.

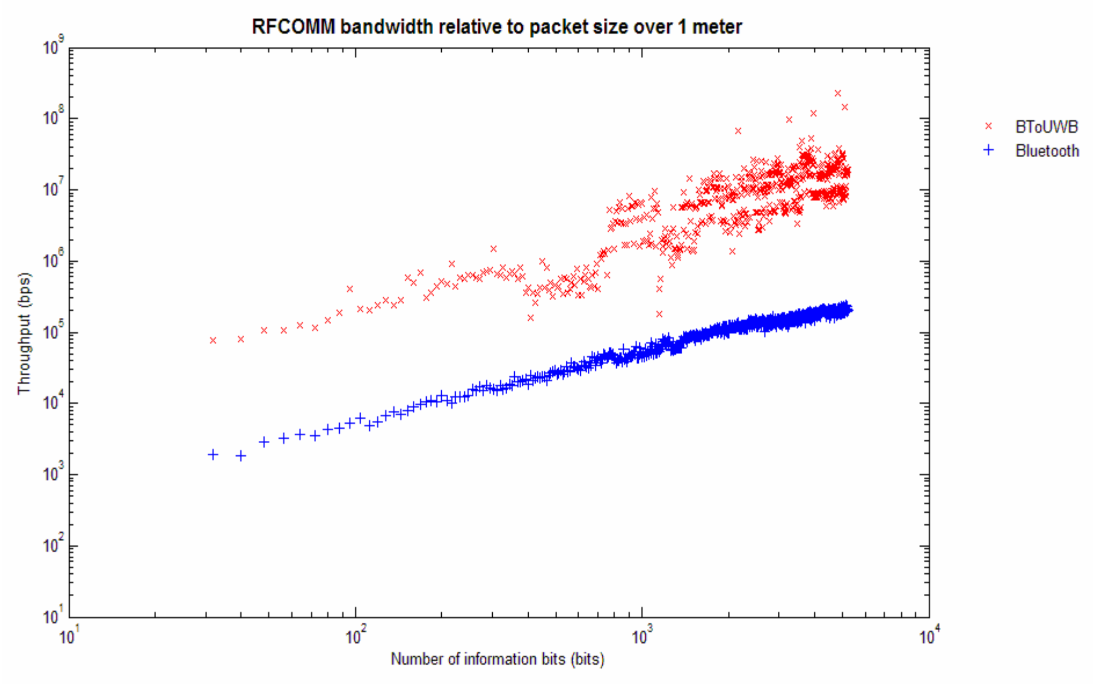

Figure 26: RFCOMM data throughput over 1 m transmission distance

At 1 and 2 meters, depicted in Figure 26 and Figure 27 respectively, similar bandwidth improvements are observed for the BToUWB system. The detrimental effect that the increased transmission distance is starting to have on both the Bluetooth and BToUWB systems can also be seen in Figure 27.

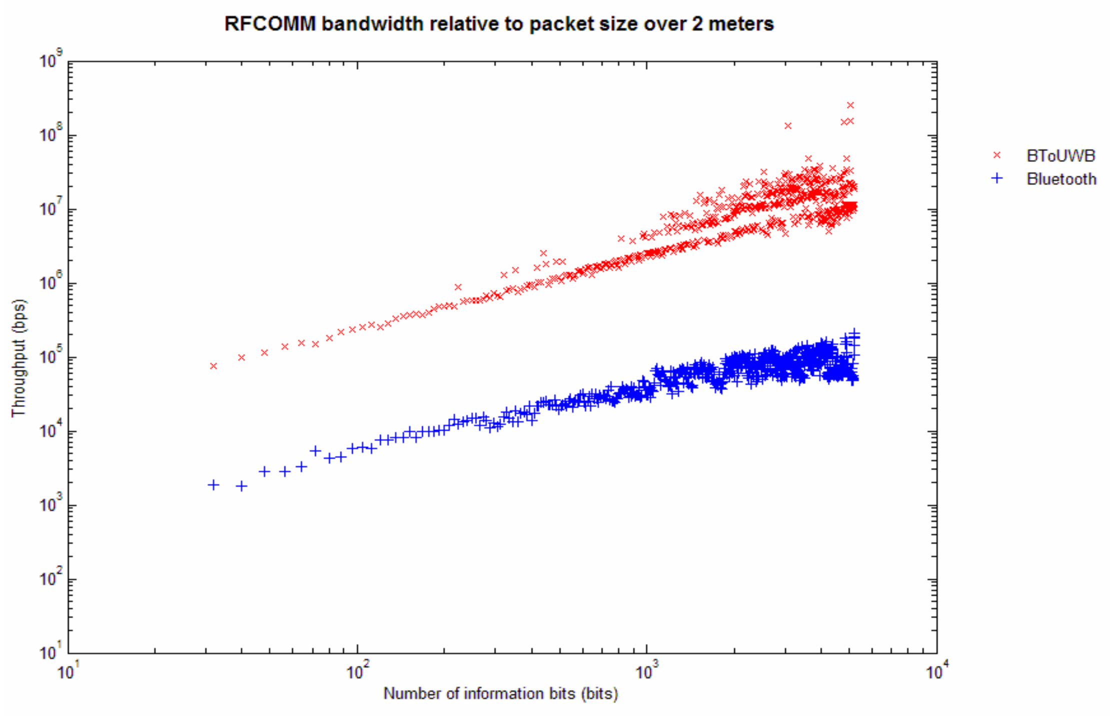

Figure 27: RFCOMM data throughput over 2 m transmission distance

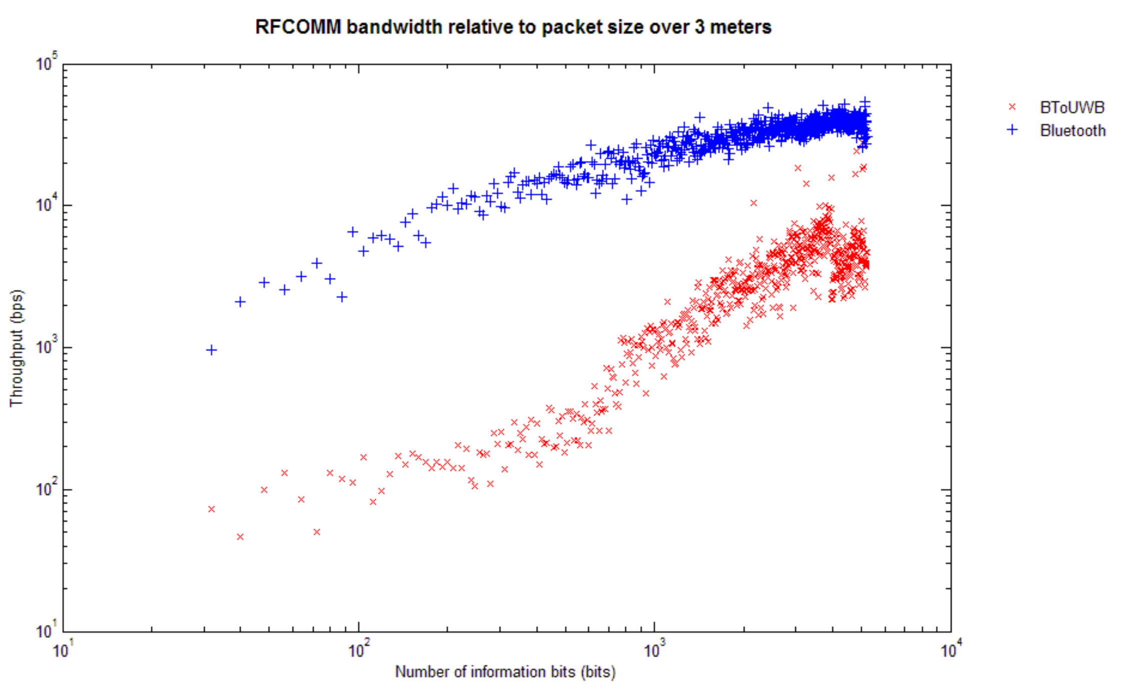

Figure 28: RFCOMM data throughput over 3 m transmission distance

Once the transmission distance were increased to 3 meters, both the BToUWB and Bluetooth channels started to decay in quality, with very bad performance around 3600 bits, as shown in Figure 28. The reason for this loss of bandwidth is explored in section 4.2.1.6.

4.2.1.4 RFCOMM Data: Bandwidth vs Transmission Distance

Figure 29, Figure 30 and Figure 31 illustrate the RFCOMM data throughput (bandwidth) as a function of the distance between the wireless transceivers for small, medium and large RFCOMM packet sizes respectively.

Figure 29 shows how only a slight decay in Bluetooth bandwidth (at this scale) can be observed over 0 to 3 meter transmission distance. A fairly constant throughput were obtained for 0 to 2 meter BToUWB transmissions, with a sharp decay in throughput above 2 meters, as also seen in Figure 29, where this underperformance are described in paragraph 4.2.1.6.

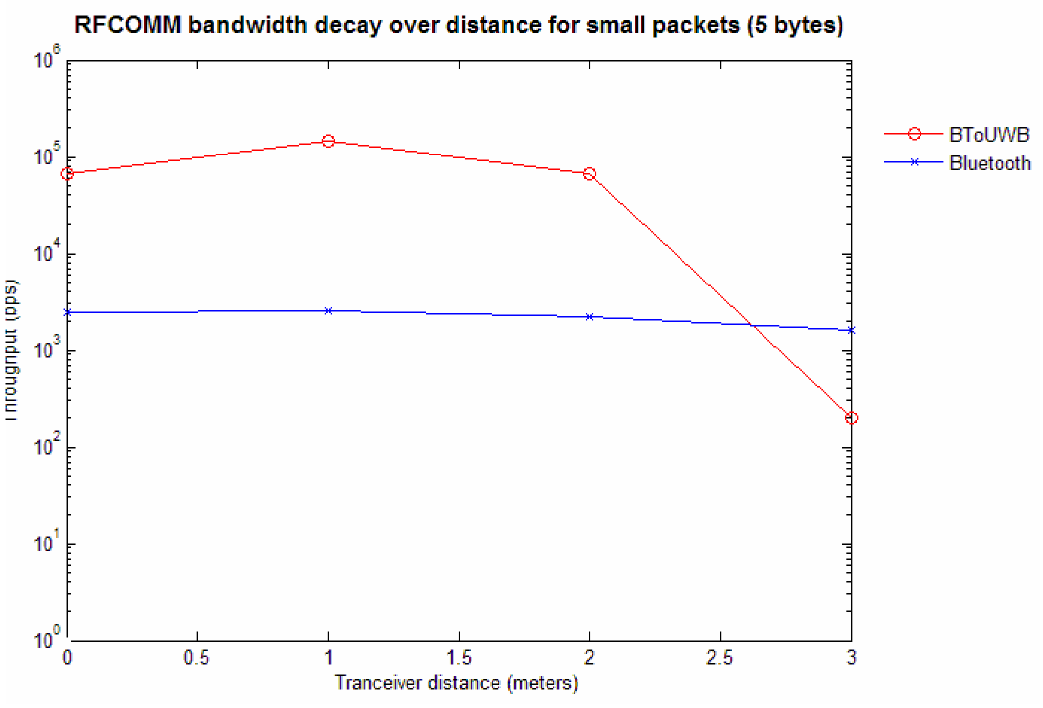

Figure 29: RFCOMM data throughput vs. transmission distance for small packets

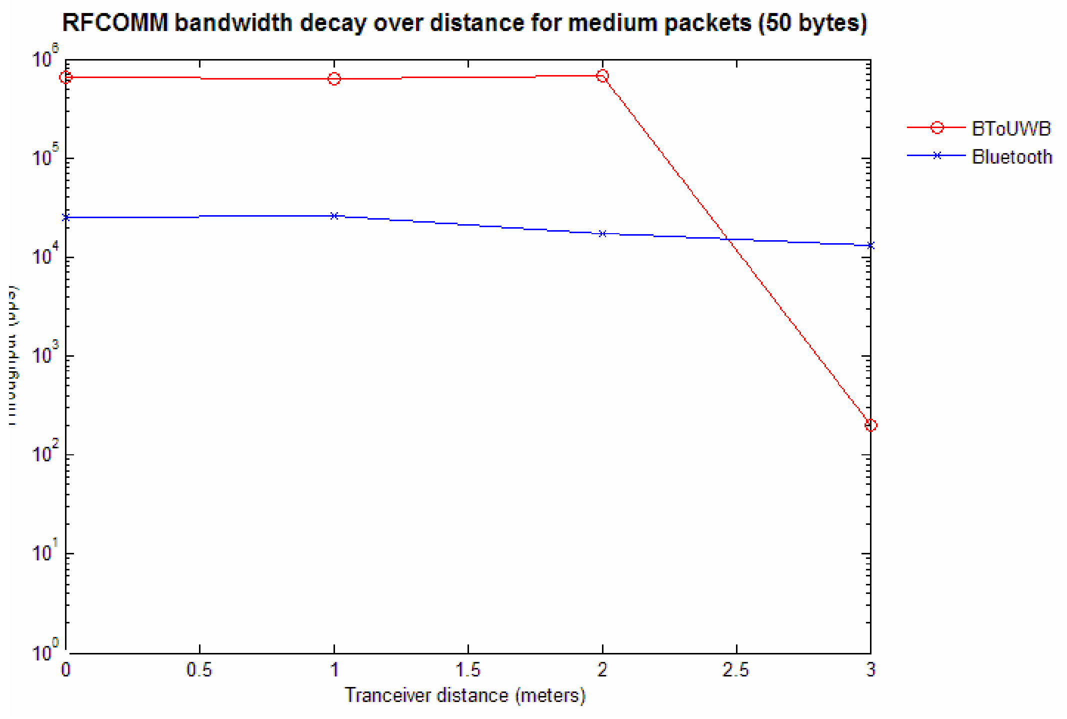

Figure 30: RFCOMM data throughput vs. transmission distance for medium sized packets

Similar to Figure 29, Figure 30 and Figure 31 shows a sharp decay of bandwidth for transmission distances above 2 meters. Section 4.2.1.6 attempts to explain this bandwidth loss.

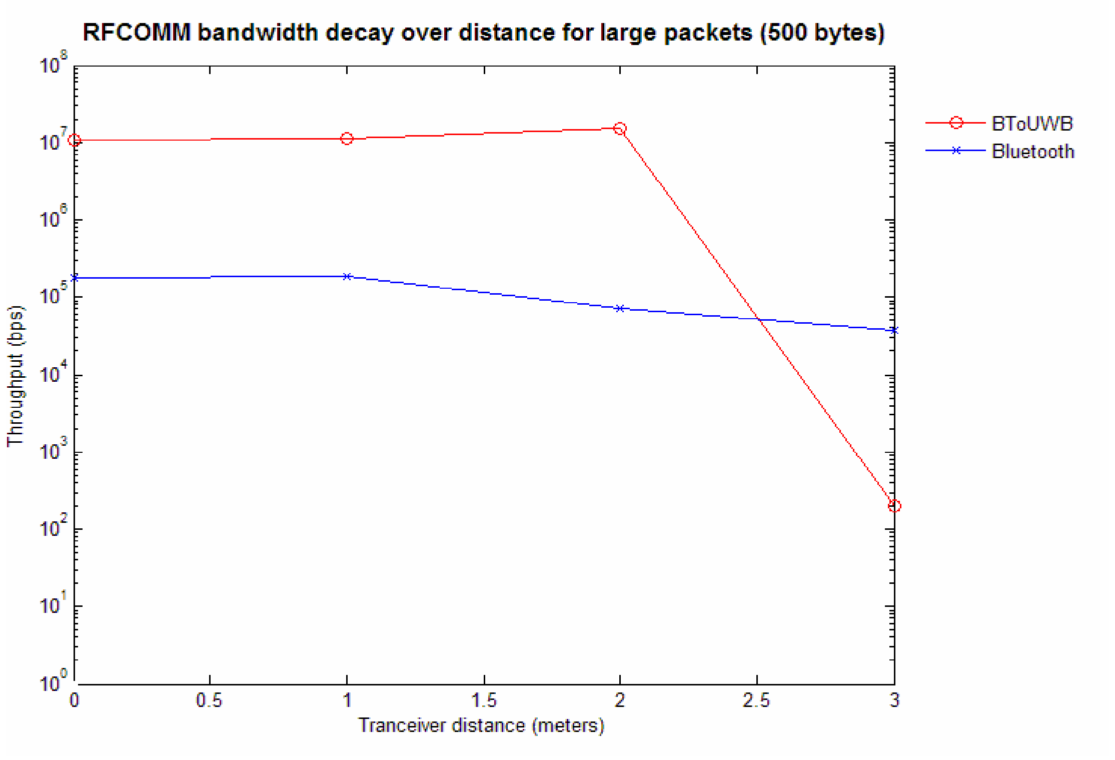

Figure 31: RFCOMM data throughput vs. transmission distance for large packets

4.2.1.5 Data Discontinuity

This section attempts to explain the L2CAP ACL throughput discontinuity observed around 3500 information bits in Figure 18 to Figure 20.

In the header file `i1480u-wlp.h` of the applied Linux UWB v1.8.9 driver [35], the following statement explains the limitation put on packet sizes transmitted over the UWB subsystem:

```
UNTD packets (UNTD hdr + i1480 hdr + network packet) packets
cannot be bigger than i1480u_MAX_FRG_SIZE. When this happens, the
i1480 packet is broken in chunks/packets:

UNTD-1st.hdr + i1480.hdr + payload
UNTD-next.hdr + payload
UNTD-last.hdr + payload

so that each packet is smaller or equal than i1480u_MAX_FRG_SIZE.
```

In the same file the following definitions are found:

```
i1480u_MAX_RX_PKT_SIZE = 4114,
i1480u_MAX_FRG_SIZE = 512,
i1480u_RX_BUFS = 9,
```

This means that packets are fragmented into 512 byte chunks to be transmitted over the UWB subsystem.

Transmitting approximately 3500 Bluetooth application information bits over the BToUWB implemented system, relates to 3500 / 8 ≈ 440 bytes. Adding another 52 overhead bytes, as stipulated in Table 24, together with unaccounted overhead and padding for alignment, will result in excess of 512 byte packets requiring fragmentation when exceeding 3500 information bits.

Level	***Defined in***	***Number of overhead bytes***
L2CAP	`l2cap.h`	4
L2CAP external routing	`l2cap.h`	16
UWB Router	`uwb_router_src.h`	4
WLP Conversion / Network	`socket.h`	16
WLP	`i1480-wlp.h`	12
Total		52

Table 24: Packet overhead byte allocation

The processing time to perform this added overhead on all MAC level packets at both transmitting and receiving ends twice for loop-backed packets will invariable have a negative impact on the data throughput, hence the discontinuity in BToUWB data observed in Figure 18 to Figure 20.

4.2.1.6 Bandwidth Decay

Figure 21 through Figure 24 and Figure 28 through Figure 31 all show a dramatic loss of bandwidth for the BToUWB system at transmission distances above 2 meters. Although the UWB dongles' specification stated "connections up to 10 meters in open space", the BToUWB system unexpectedly performed much worse that the Bluetooth system overall at 3 meters. This contradicting poor UWB performance over longer distances may be attributed to ineffective or faulty introductory hardware or the inefficient unreleased WLP firmware obtained from Intel Corporation without any guarantees.

4.2.1.7 Theoretical Application Throughput

For comparison, a theoretically expected BToUWB throughput were added to the graphs in Figure 22 through Figure 24 for packet sizes 5, 50 and 500 respectively. This section shows how the theoretical expected value were calculated for each of these packet sizes.

If we assume that the radio channel is the bottle neck in the system, we can calculate the total channel throughput as:

$$Throughput_{Total} = \frac{Information\_Bits_{Total}}{Transmission\_Time_{Radio}} \qquad (5)$$

$$\therefore Transmission\_Time_{Radio} = \frac{Information\_Bits_{Total}}{Throughput_{Total}} \tag{6}$$

$$= \frac{Information\_Bits_{Application} + Information\_Bits_{Overhead}}{Throughput_{Total}} \tag{7}$$

The actual data throughput seen by an upper layer Bluetooth application will therefore be:

$$Throughput_{Application} = \frac{Information\_Bits_{Application}}{Transmission\_Time_{Radio} + Transmission\_Time_{Other}} \tag{8}$$

If we assume that the time it takes for the data to progress through the rest of the channel can be estimated as 80% of the time it takes to progress through the radio channel for high speed UWB, we can rewrite equation (8) as:

$$Throughput_{Application} = \frac{Information\_Bits_{Application}}{1.8 \times Transmission\_Time_{Radio}} \tag{9}$$

Substituting equation (7) into (9) we obtain the following:

$$Throughput_{Application} = \frac{Information\_Bits_{Application} \times Throughput_{Total}}{1.8 \times \left(Information\_Bits_{Application} + Information\_Bits_{Overhead}\right)} \tag{10}$$

If we assume a 200 Mbps Total Channel Throughput for the UWB radio channel, as advertised, and assume 52 bytes to be the number of overhead information bytes, as described in section 4.2.1.5, resulting in 52 x 8 = 416 *Overhead Information Bits*, we can calculate the theoretical data throughput that should be seen by an upper layer Bluetooth application using the BToUWB system for each of the investigated packet sizes, as shown in Table 25.

Packet Size	Application Information Bits	Total Throughput (Mbps)	Overhead Information Bits	Application Throughput (Mbps)
5	40	200	416	9.75
50	400	200	416	54.47
500	4000	200	416	100.64

Table 25: Application Throughput Calculation

This computation ignores any other channel effects like interference, delay, path loss or fading.

4.2.2 Bluetooth Isochronous Data Analysis

The results obtained from analysing the Bluetooth and BToUWB SCO channels, as described in section 3.3.2, are portrayed in Figure 32 and Figure 33. More specifically, Figure 32 shows the Peak Signal-to-Noise Ratio (PSNR) of the left and right channel received audio signal against the transmitted audio signal for both the Bluetooth and BToUWB systems, where a low quality (8kHz stereo 16-bit audio sample rate) PCM audio signal were used as input.

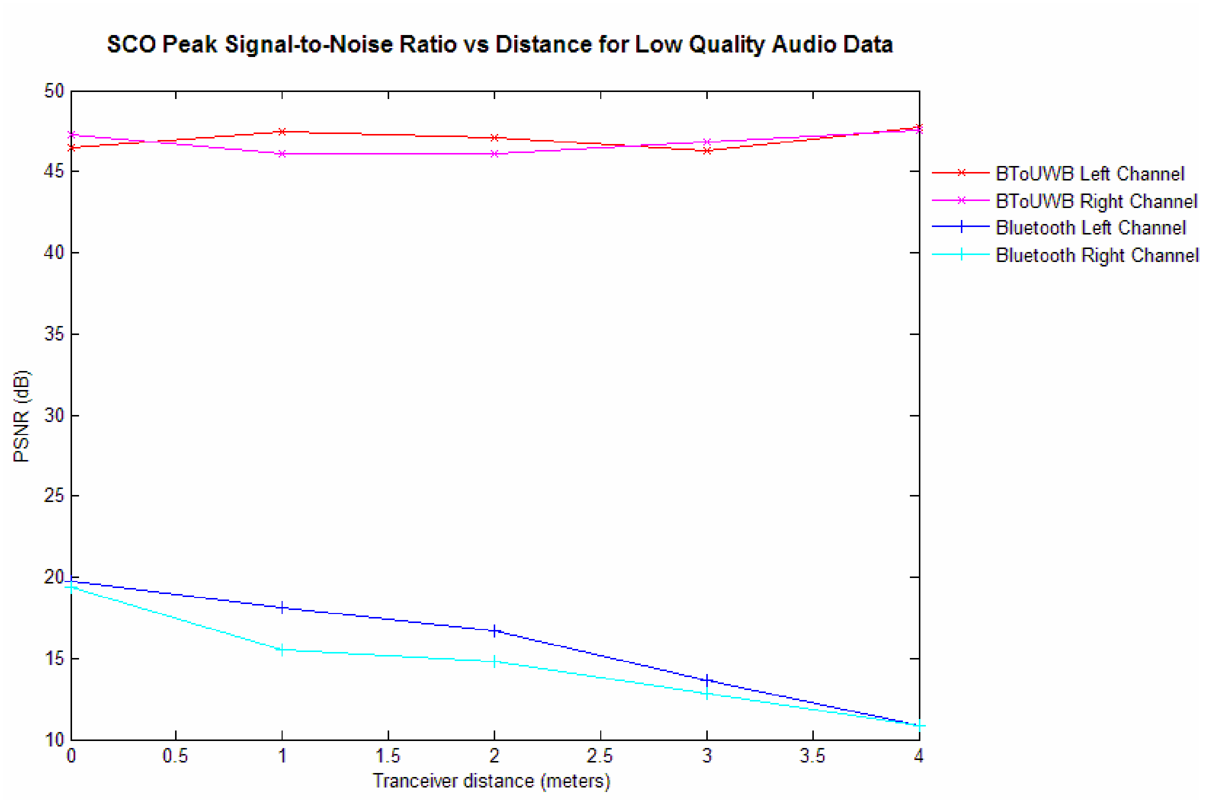

Figure 32: SCO PSNR vs. transmission distance for low quality audio data

Figure 32 shows how the implemented BToUWB system provided drastically improved SCO channel quality. Whilst the Bluetooth SCO channel quality clearly deteriorates as the transmission distance increases, the BToUWB SCO remain constantly high.

Figure 33 shows the Peak Signal-to-Noise Ratio (PSNR) of the left and right channel received audio signal against the transmitted audio signal for both the Bluetooth and BToUWB systems, where a high quality (44kHz stereo 16-bit audio sample rate) PCM audio signal were used as input.

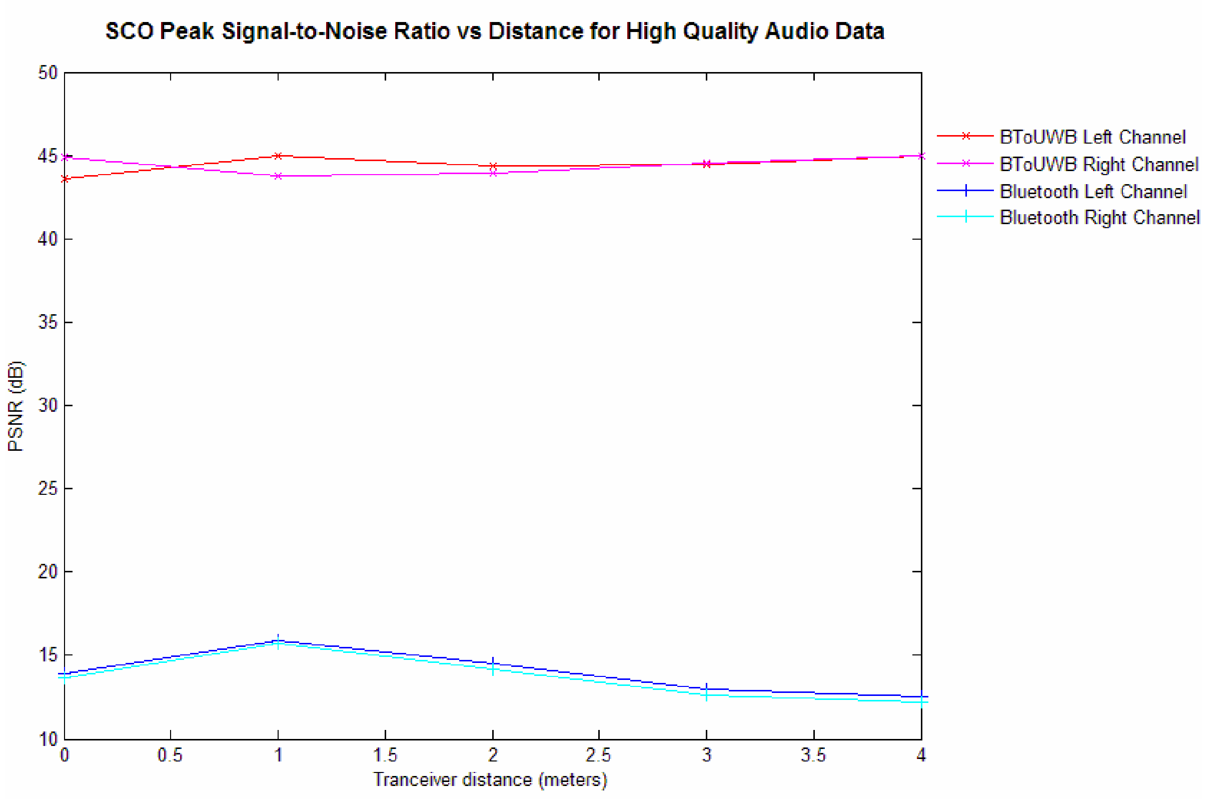

Figure 33: SCO PSNR vs. transmission distance for high quality audio data

As in Figure 32, Figure 33 shows how the implemented BToUWB system provided much improved SCO channel quality constantly over various transmission distances.

CHAPTER 5: CONCLUSION

5.1 SUMMARY

A BToUWB system were reliably simulated and successfully implemented with Bluetooth and UWB hardware, showing that UWB can indeed be implemented as lower layer radio communication for Bluetooth enabled devices. The system enabled the transmission and analyses of Bluetooth asynchronous L2CAP and RFCOMM data and synchronous SCO data over an Ultra Wideband link. The system also proved the HCI layer to be a reliable mergence point between the Bluetooth and UWB protocols.

The obtained results show that the Bluetooth connection achieved a 50 fold increase in bandwidth when tunnelling L2CAP or RFCOMM ACL data over a UWB radio link for various packet sizes and transmission distances of 0 to 2 meters. In theory a much bigger bandwidth gain should be expected from UWB. This mismatch and the rapid decay in bandwidth above 2 meters may be due to defects in the pilot UWB radio hardware and firmware implemented for the developed BToUWB system, or insufficient dataflow optimisation in the Linux UWB driver. A selection policy for this system would then be to transmit ACL data over the UWB link for distances less than 2 meters. The dramatic quality increase obtained for synchronous SCO data is due to the high speed channelling of synchronous data over a loss-less asynchronous UWB connection. Despite the increased bandwidth, the UWB system used in this implementation did not put any advantages forward in terms of transmission distances.

The rest of this chapter provides a further in-depth discussion of the results obtained in Chapter 4.

5.2 BREAKDOWN

5.2.1 Bluetooth over UWB Simulation

Assuming that the simulated UWB link may closely resemble a real life UWB connection, the obtained results show that a Bluetooth connection will benefit from a UWB physical transmission layer for all ACL packet sizes. The system design showed that UWB mergence into the Bluetooth stack is indeed possible at the HCI layer, and with a true UWB radio controller this system may be re-applied to obtain even more realistic measurements.

The simulation proposed in section 3.1.1 contains some inherent shortfalls. Using a simulated UWB radio link in place of a real UWB implementation poses a number of limitations on the undertaken research and results obtained, including insufficient real life imitation of interference, true transfer speeds and most of all the effects of distance between transceivers. The encoding, medium contention, modulation and frequency spreading operations performed by the physical layer were also not simulated.

5.2.2 Bluetooth over UWB System Implementation

5.2.2.1 L2CAP Asynchronous Data

The results obtained in section 4.2.1.1 shows that the Bluetooth protocol stack will indeed benefit in terms of bandwidth when tunnelling L2CAP ACL data over a UWB radio link instead of a Bluetooth radio link for various packet sizes and transmission distances of 0 to 2 meters. The measurements recorded a 50 fold increase of bandwidth when using the UWB radio link. However, at a wireless transmission distances above 2 meters, the implemented UWB link quality rapidly decays against expectations to less than that of the Bluetooth link. The overall bandwidth for distances less than 2 meters is also more or less 100 times less than the theoretical expected rate. This may be due to defects in the pilot UWB radio hardware and pre-released Intel i1480 firmware implemented for the developed BToUWB system, and also insufficient dataflow optimisation in the Linux UWB driver. At a transmission distance of 3 meters, both the Bluetooth and BToUWB links suffered severe quality impairments, especially with increased packet sizes.

The results obtained from the developed BToUWB system proposes a selection policy for L2CAP ACL data to only be transmitted over the UWB link for transmission distances less than 2 meters, requiring employment of the position measurement functionality offered by UWB technology.

5.2.2.2 RFCOMM Asynchronous Data

Due to the fact that the Bluetooth RFCOMM layer implements L2CAP ACL data channels for communication, the results measured for RFCOMM data analysis in section 4.2.1.3 is expected to coincide with that obtained for L2CAP analysis in section 4.2.1.1. We see that the RFCOMM bandwidth indicators indeed follow the same trends as for L2CAP analysis, with the RFCOMM data throughput approximately 50 times higher for the BToUWB link than for the vanilla Bluetooth link, with a rapid bandwidth decay recorded for transmission distances above 2 meters.

The results obtained from the developed BToUWB system also suggests a selection policy for RFCOMM data to only be transmitted over the UWB link for transmission distances less than 2 meters. This would however automatically be embedded into the lower L2CAP layer.

5.2.2.3 SCO Isochronous Data

Section 3.2.2.1 describes the scheme chosen to improve the Bluetooth SCO channel by applying the advantageous higher bandwidth offered by UWB. Figure 32 and Figure 33 in section 4.2.2 shows a dramatic improvement in the SCO channel quality for audio data transmitted over the developed BToUWB link compared to that transmitted over a traditional Bluetooth link. From a system design view point, it is in fact expected that the BToUWB SCO data should not bare any quality loss at all, whereas the measurements portrayed in the two graphs shows a minimal loss. This can be owing to a cut-off from the tail end of the received data samples due to the timing offset between the transmitted and received data.

Overall the SCO channel quality was improved to a maximum when implementing the BToUWB scheme for transmitting and buffering SCO data.

5.2.3 Bluetooth mergence with UWB

The developed BToUWB system proved the Bluetooth HCI layer to be a viable point of mergence with the UWB lower layer stack. The system also showed a bandwidth improvement for both L2CAP and RFCOMM connections when implementing the UWB radio link for transmissions over short distances, whilst an improvement in channel quality can be achieved for SCO data when applying the bandwidth advantages posed by a UWB link. Despite the increased bandwidth, the UWB system used in this implementation did not put any advantages forward in terms of transmission distances.

Discovery of nearby devices and establishment of a primary connection between two wireless devices were solely performed by the traditional proven Bluetooth methods and Bluetooth radio subsystem. Utilising the UWB physical layer for some or all of these tasks may provide even more advantages in terms of connection setup speed and device power saving.

UWB connection setup and hibernation may be left to be completely handled by the UWB subsystem, or enabled under Bluetooth control upon a UWB service discovery. Power requirements and connection setup and wakeup speed will all be factors to consider upon investigating an optimal solution for these challenges.

5.3 FUTURE CONTRIBUTIONS

The comprehensive scope of the project and lack of ample accessibility to UWB hardware and firmware limited the analysis performed to the various data transmissions offered by the Bluetooth protocol between two nodes with an existing Bluetooth link.

To extend the research the developed system can be expanded to:

- Perform connection setup tasks over UWB instead of Bluetooth. These include:
  - Device discovery
  - Synchronisation
  - Authentication
  - Other L2CAP functionality like grouping and information (see section 2.1.1.5)

- Implement flow control for the proposed SCO over UWB system
- Investigate UWB isochronous medium access for the transmission of SCO data
- Bypass the UWB WLP overhead with a direct interface to the UWB radio controller via the USB channel
- Setup a multi-node network and analyse its performance impact

REFERENCES

[1] T. Rappaport, *Wireless Communications: Principles and Practice*, Prentice Hall, Pages 52 - 53, January 2002.

[2] N.J. Muller, *Bluetooth Demystified*, McGraw-Hill Telecom, 2001.

[3] IEEE Std. 802.11, *IEEE Standard for Wireless LAN Medium Access Control (MAC) and Physical-Layer Specification*, November 1997.

[4] Bluetooth Core Specification, http://www.bluetooth.com, 2004.

[5] J.M. Wilson, "Ultra Wideband, a Disruptive RF technology?", *Intel Research and Development*, August 2002.

[6] Anon., "Ultrawide Boys Hit The Radio Spectrum", *Electronics Systems & Software*, Volume 1 Issue 5, p40, October 2003.

[7] D. Jackson, "Ultrawideband may thwart 802.11 Bluetooth efforts", *Telephony*, Volume 242 Issue 6, p16, February 2002.

[8] Anon., "The Commercial Market", *Microwave Journal*, Volume 46 Issue 9, p65, September 2003.

[9] "Top Ten wireless", *Electronic Design*, Volume 52 Issue 1, p104, January 2004.

[10] S. Deffree, "Will Bluetooth Be Bumped?", *Electronic News (North America)*, Volume 49 Issue 52, December 2003.

[11] K. Mandke, H. Nam, L. Yerramneni, C. Zuniga and T Rappaport, "The Evolution of Ultra Wide Band Radio for Wireless Personal Area Networks", *High Frequency Electronics*, Summit Technical Media, Pages 22 - 32, 2003.

[12] G.R. Aiello and G.D. Rogerson, "Ultra Wideband Wireless Systems", *IEEE Microwave Magazine*, Pages 36 - 47, June 2003.

[13] D. Porcino and W. Hirt, "Ultra Wideband Radio Technology: Potential and Challenges Ahead", *IEEE Communications Magazine*, Pages 66 - 74, July 2003.

[14] D.G. Leeper, "Ultrawideband - the next step in short-range wireless", *IEEE Symposium on Radio Frequency Integrated Circuits*, Pages 493 - 496, June 2003.

[15] D. Johnson (djohnson@csir.co.za), Thoughts on proposed dissertation: An investigation into the viability of UWB as lower-layer for Bluetooth, e-mail to G.P. Hancke (gerhard.hancke@eng.up.ac.za), 22 April 2004.

[16] Palo Wireless, "Bluetooth Resource Center - Bluetooth Tutorial", *Palo Wireless Online Website*, http://www.palowireless.com/infotooth/tutorial.asp, November 2007.

[17] Bluetooth SIG, "Bluetooth Baseband Specification", p42, 2007.

[18] BlueZ Project, "Official Linux Bluetooth protocol stack", *BlueZ Online Website*, http://www.bluez.org/, November 2007.

[19] GNOME Bluetooth, "About the GNOME Bluetooth subsystem", *Gnome Bluetooth Online Website*, http://live.gnome.org/GnomeBluetooth, November 2007.

[20] Gentoo Linux Documentation, "Gentoo Linux Bluetooth Guide", *Gentoo Online Website*, http://www.gentoo.org/doc/en/bluetooth-guide.xml, November 2007.

[21] J. Beutel and M. Krasnyanskly, "Bluetooth Protocol Stack for Linux - Linux BlueZ Howto", *Jeremy Thompson's Personal Online Website*, http://www.jeremythompson.uklinux.net/RH8-0/bluezhowto.pdf, 14 November 2001.

[22] Hans-Cees Speel, "BlueZ User Howto", Ver. 0.6, *Hans-Cees Speel's Personal Online Website*, http://www.hanscees.com/bluezhowto.html, 20 April 2003.

[23] I. Pérez-González and R. Chatré, "Linux UWB + Wireless USB + WiNET", *Linux UWB Online Community*, http://www.linuxuwb.org/thewiki/Linux-UWB+WUSB+WiNET, 13 January 2008.

[24] Intel Corporation, "Intel Wireless UWB Link 1480 Ultra Wideband MAC", *Intel Online Website*, http://www.intel.com/network/connectivity/products/uwb/index.htm, 13 January 2008.

[25] WiMedia Alliance, *Online Community Website*, http://www.wimedia.org, April 2007.

[26] Standard ECMA-368, "High Rate Ultra Wideband PHY and MAC Standard", 2nd edition, *ECMA International Standards*, http://www.ecma-international.org/publications/standards/Ecma-368.htm, December 2007.

[27] Standard ECMA-369, "MAC-PHY Interface for ECMA-368", 2nd edition, *ECMA International Standards*, http://www.ecma-international.org/publications/standards/Ecma-369.htm, December 2007.

[28] M. Hämäläinen, J. Saloranta, J. Mäkelä, I. Oppermann and T. Patana, "Ultra Wideband Signal Impact on IEEE802.11b and Bluetooth Performances", *IEEE Int. Symp. on Personal, Indoor and Mobile Radio Communications*, PIMRC '03, pp. 2943 - 1947, 2003.

[29] A. Palin, J. Reunamäki, J. Salokannel and J. Ylänen, "Bluetooth Host Protocol Usage Over the Ultra Wideband Radio", *IEEE Int. Symp. on Personal, Indoor and Mobile Radio Communications*, PIMRC '06, pp. 1 - 4, September 2006.

[30] Albert S. Huang, “The Use of Bluetooth in Linux and Location Aware Computing”, *Master's thesis*, Massachusetts Institute of Technology, Cambridge, 2005.

[31] Albert S. Huang, “An Introduction to Bluetooth Programming”, *Online Website*, http://people.csail.mit.edu/albert/bluez-intro/, September 2007.

[32] Mercurial Source Control Management System, *Online Wiki*, http://www.selenic.com/mercurial/wiki/, February 2008.

[33] Open Source SuSE Linux, Novell Inc, 2008: *Online Community Website*, http://www.opensuse.org, August 2007.

[34] Falko Timme, “How to compile a kernel – The SuSE way”, *HowtoForge Online Community Website*, http://www.howtoforge.com/kernel_compilation_suse, 17 November 2006.

[35] I. Pérez-González, “Linux UWB v1.8.9 Release Notes”, *Linux UWB Online Community*, http://www.linuxuwb.org/thewiki/README-1.8.9, 30 April 2007.

[36] “GUWA100U Wireless USB Host Adapter”, *Online Product Description*, IOGear Inc., http://www.iogear.com/product/GUWA100U, March 2008.

[37] Chris Bagwell, “SoX - Sound eXchange”, *Online Website*, http://sox.sourceforge.net/, May 2008.

ADDENDUMS

A BLUEZ AND GENERIC UTILITIES

A.1 HCICONFIG

A.1.1 Description

HCICONFIG is an HCI device configuration utility

A.1.2 Usage

	Device	Command	Description
`hciconfig`	`hciX`	`up`	Open and initialise HCI device
		`down`	Close HCI device
		`reset`	Reset HCI device
		`rstat`	Reset stat counters
		`auth`	Enable Authentication
		`noauth`	Disable Authentication
		`encrypt`	Enable Encryption
		`noencrypt`	Disable Encryption
		`secmgr`	Enable Security Manager
		`nosecmgr`	Disable Security Manager
		`piscan`	Set page scan and inquiry scan mode
		`noscan`	Disable scan modes
		`iscan`	Set inquiry scan mode only
		`pscan`	Set page scan mode only
		`ptype [type]`	Get/Set default packet type
		`lm [mode]`	Get/Set default link mode
		`lp [policy]`	Get/Set default link policy
		`name [name]`	Get/Set local name
		`class [class]`	Get/Set class of device
		`voice [voice]`	Get/Set voice setting
		`iac [iac]`	Get/Set inquiry access code
		`inqmode [mode]`	Get/Set inquiry mode
		`inqdata [data]`	Get/Set inquiry data
		`inqtype [type]`	Get/Set inquiry scan type
		`inqparms [win:int]`	Get/Set inquiry scan window and interval
		`pageparms [win:int]`	Get/Set page scan window and interval
		`pageto [to]`	Get/Set page timeout
		`afhmode [mode]`	Get/Set AFH mode
		`aclmtu <mtu:pkt>`	Set ACL MTU and number of packets
		`scomtu <mtu:pkt>`	Set SCO MTU and number of packets
		`putkey <bdaddr>`	Store link key on the device
		`delkey <bdaddr>`	Delete link key from the device
		`commands`	Display supported commands
		`conn`	Show active connections
		`features`	Show features
		`version`	Display version information
		`revision`	Display revision information

A.1.3 Examples

To query the current default packet type:

```
hciconfig hci0 ptype
```

To set the new packet type:

```
hciconfig hci0 ptype <types list separated by comma>
```

A.2 HCIATTACH

A.2.1 Description

HCIATTACH is an HCI UART driver utility that initialises a given serial port. Using this tool you can add/delete UART devices without restarting HCId.

A.2.2 Usage

	Interface	Type	Options

`hciattach`	`<tty>`	`<type | id>`	`[speed]`
			`[flow|noflow]`
			`[-s initial_speed]`
			`[-t timeout]`
			`[-n]`
			`[-p]`
			`[-b]`
			`-l`

A.2.3 Examples

```
hciattach ttyS0 xircom 115200 flow
hciattach ttyS1 ericsson 115200 flow
hciattach ttyS2 any 57600
```

A.3 HCITOOL

A.3.1 Description

HCITOOL is used for generic writing to and monitoring of the HCI interface.

A.3.2 Usage

A.3.2.1 HCI Monitoring

```
hcitool [options] <command> [command parameters]
```

	Device	Command	Description
`hcitool`	`[-i hciX]`	`dev`	Display local devices
		`inq`	Inquire remote devices
		`scan`	Scan for remote devices
		`name`	Get name from remote device
		`info`	Get information from remote device
		`spinq`	Start periodic inquiry
		`epinq`	Exit periodic inquiry
		`cmd`	Submit arbitrary HCI commands
		`con`	Display active connections
		`cc`	Create connection to remote device
		`dc`	Disconnect from remote device
		`sr`	Switch master/slave role
		`cpt`	Change connection packet type
		`rssi`	Display connection RSSI
		`lq`	Display link quality
		`tpl`	Display transmit power level
		`afh`	Display AFH channel map
		`lst`	Set/display link supervision timeout
		`auth`	Request authentication
		`enc`	Set connection encryption
		`key`	Change connection link key
		`clkoff`	Read clock offset
		`clock`	Read local or remote clock

A.3.2.2 HCI Writing

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

`OGF`	`OpCode Group Field (00-3F)`
`OCF`	`OpCode Command Field (0000-03FF)`
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.`

A.3.3 Examples

The following example shows how to create a Bluetooth link between two devices:

```
hcitool cc 00:0A:94:00:1A:EC
```

And then testing the link quality of the link:

```
hcitool lq 00:0A:94:00:1A:EC
Link quality: 255
```

An example to do an inquiry using LAP 0x9E8B33 for 10 x 1.28 sec and unlimited responses is:

```
hcitool -i hci0 01 0001 33 8b 9e 10 00
```

and to stop the inquiry:

```
hcitool -i hci0 01 0002
```

A.4 HCIDUMP

A.4.1 Description

HCIDUMP is an HCI packet analyser

A.4.2 Usage

```
hcidump [OPTION...] [filter]
```

Option	Alternative	Description
`-i,`	`--device=hci_dev`	HCI device
`-l,`	`--snap-len=len`	Snap len (in bytes)
`-p,`	`--psm=psm`	Default PSM
`-m,`	`--manufacturer=compid`	Default manufacturer
`-w,`	`--save-dump=file`	Save dump to a file
`-r,`	`--read-dump=file`	Read dump from a file
`-s,`	`--send-dump=host`	Send dump to a host
`-n,`	`--recv-dump=host`	Receive dump on a host
`-t,`	`--ts`	Display time stamps
`-a,`	`--ascii`	Dump data in ascii
`-x,`	`--hex`	Dump data in hex
`-X,`	`--ext`	Dump data in hex and ascii
`-R,`	`--raw`	Dump raw data
`-C,`	`--cmtp=psm`	PSM for CMTP
`-H,`	`--hcrp=psm`	PSM for HCRP
`-O,`	`--obex=channel`	Channel for OBEX
`-P,`	`--ppp=channel`	Channel for PPP
`-D,`	`--pppdump=file`	Extract PPP traffic
`-A,`	`--audio=file`	Extract SCO audio data
`-B,`	`--btsnoop`	Use BTSnoop file format
`-V,`	`--verbose`	Verbose decoding

`-Y,`	`--novendor`	No vendor commands or events
`-N,`	`--noappend`	No appending to existing files
`-4,`	`--ipv4`	Use IPv4 as transport
`-6`	`--ipv6`	Use IPv6 as transport
`-h,`	`--help`	Give this help list
	`--usage`	Give a short usage message

A.4.3 Example

Following is an example of an hcidump command with expected output:

```
hcidump –I hci0
HCI sniffer - Bluetooth packet analyser ver 1.33
device: hci0 snap_len: 1028 filter: 0xffffffff
> HCI Event: Number of Completed Packets (0x13) plen 5
> ACL data: handle 41 flags 0x02 dlen 52
    L2CAP(s): Echo rsp: dlen 44
< HCI Command: Disconnect (0x01|0x0006) plen 3
```

A.5 HCIEMU

A.5.1 Description

HCIEMU is an HCI emulation daemon

A.5.2 Usage

```
hciemu [-n] local_address
```

A.5.3 Example

Creating an emulated HCI device can be achieved by a command such as:

```
hciemu 00:0A:94:00:1A:D1
```

A.6 L2PING

A.6.1 Description

L2PING is a pinging utility for testing L2CAP connectivity

A.6.2 Usage

	Options	**Target**
`l2ping`	`[-s size]`	`<bd_address>`
	`[-c count]`	
	`[-t timeout]`	

```
                        [-S source_address]
                        [-f]
                        [-r]
```

A.6.3 Example

```
l2ping -s 150 -c 70 00:0A:94:00:1A:EC
Ping: 00:0A:94:00:1A:EC from 00:0A:94:00:1A:D6 (data size 150) ...
150 bytes from 00:0A:94:00:1A:EC id 0 time 77.31ms
150 bytes from 00:0A:94:00:1A:EC id 1 time 37.38ms
150 bytes from 00:0A:94:00:1A:EC id 2 time 45.28ms
...
```

A.7 L2TEST

A.7.1 Description

L2TEST is a debug utility for testing L2CAP channel functionality and throughput.

A.7.2 Usage

```
l2test <mode> [options] [bdaddr]
```

Modes:	`-r`	Listen and receive
	`-w`	Listen and send
	`-d`	listen and dump incoming data
	`-x`	Listen, then send, then dump incoming data
	`-s`	Connect and send
	`-u`	Connect and receive
	`-y`	Connect and be silent
	`-c`	Connect / Disconnect
	`-m`	Multiple connects
	`-z`	Information request
Options:	`-I imtu`	Incoming MTU that we accept
	`-O omtu`	Minimum outgoing MTU that we need
	`-b bytes`	Size of the data chunks in bytes
	`-P psm`	Use this PSM
	`-S src_addr`	Use this source address
	`-i device`	Use this device
	`-L seconds`	enable SO_LINGER
	`-B filename`	use data packets from file
	`-N num`	send num frames (default = infinite)
	`-C num`	send num frames before delay (default = 1)
	`-D ms`	delay after sending num frames (default = 0)
	`-R`	reliable mode

`-G`	use connectionless channel (datagram)
`-F`	enable flow control
`-A`	request authentication
`-E`	request encryption
`-S`	secure connection
`-M`	become master

A.7.3 Examples

Server: `l2test -I 2000 -r`

Client: `l2test -O 2000 -s <bd_addr>`

A.8 SCOTEST

A.8.1 Description

SCOTEST is a debug utility for testing SCO channel functionality and throughput.

A.8.2 Usage

```
scotest <mode> [-b bytes] [bd_addr]
```

Modes:	`-d`	`Dump (server)`
	`-c`	`Reconnect (client)`
	`-m`	`Multiple connects (client)`
	`-r`	`Receive (server)`
	`-s`	`Send (client)`

A.9 RFCOMM

A.9.1 Description

RFCOMM is used establish a “Radio Frequency Connection” to another Bluetooth device with RFCOMM available.

A.9.2 Usage

```
rfcomm [options] <command> <dev>
```

Modes:	`-i [hciX|bdaddr]`	Local HCI device or BD Address
	`-h, --help`	Display help
	`-r, --raw`	Switch TTY into raw mode
	`-f, --config [file]`	Specify alternate config file
	`-a`	Show all devices (default)

Options:	`bind`	`<dev> <bdaddr> [channel]`	Bind device
	`release`	`<dev>`	Release device

`show`	`<dev>`	Show device
`connect`	`<dev> <bdaddr> [channel]`	Connect device
`listen`	`<dev> [channel]`	Listen

A.9.3 Example

On the client side, use the command

```
rfcomm connect 0 00:0A:94:00:1A:D6 2
Connected /dev/rfcomm0 to 00:0A:94:00:1A:D6 on channel 2
Press CTRL-C for hangup
```

On the server side, use

```
rfcomm listen 0 2
Waiting for connection on channel 2
Press CTRL-C for hangup
```

B BLUETOOTH PROGRAMMING

Following is a short description of common variables used with most functions provided in the Linux Bluetooth library, followed by a description of the most important functions themselves [31]. Section B.3 provides an example implementation where these programming concepts are put to use for basic Bluetooth functionality testing.

B.1 Common Bluetooth variables and data structures

This section provides a quick description of data structures most likely to be encountered whilst using the Linux Bluetooth API's for Bluetooth programming.

A Bluetooth device address consists of 6 octets (see section 2.1.1.2.2). BlueZ commonly uses the `bdaddr_t` structure to store these octets of a device address:

```
typedef struct {
      uint8_t b[6];
} __attribute__((packed)) bdaddr_t;
```

This data structure can easily be converted to and from a string format by using the BlueZ functions `ba2str()` and `str2ba()` respectively. The `bacpy()` copies an address from one structure to another.

`dev_id` stores the resource number of the local Bluetooth device, whilst `socket` represents a connection to the local Bluetooth controller and not a remote Bluetooth device.

The `inquiry_info` structure is used to contain a list of detected devices and some basic information about them, like their Bluetooth address, class, and clock offset:

```
struct inquiry_info {
      bdaddr_t bdaddr;
      __u8     pscan_rep_mode;
      __u8     pscan_period_mode;
      __u8     pscan_mode;
      __u8     dev_class[3];
      __le16   clock_offset;
} __attribute__ ((packed));
```

The `sockaddr_rc` structure is used to specify the details of an outgoing RFCOMM connection or listening socket:

```
struct sockaddr_rc {
      sa_family_t rc_family;
      bdaddr_t          rc_bdaddr;
      uint8_t           rc_channel;
};
```

The `rc_family` field specifies the addressing family of the socket, and will always be `AF_BLUETOOTH`. For outgoing connections, `rc_bdaddr` specifies the address of the remote Bluetooth device to connect to. For incoming connections, the `rc_bdaddr` specifies the local Bluetooth adapter to use, and is typically set to `BDADDR_ANY` to indicate that any local

Bluetooth adapter is acceptable. `rc_channel` specifies the port number to connect to / listen on.

The `sockaddr_l2` structure is used to specify the details of an outgoing L2CAP connection or listening socket:

```
struct sockaddr_l2 {
      sa_family_t l2_family;
      unsigned short    l2_psm;
      bdaddr_t          l2_bdaddr;
};
```

The `l2_family` field specifies the addressing family of the socket, and will always be `AF_BLUETOOTH`. For outgoing connections, `l2_bdaddr` specifies the address of the remote Bluetooth device to connect to. For incoming connections, the `l2_bdaddr` specifies the local Bluetooth adapter to use, and is typically set to `BDADDR_ANY` to indicate that any local Bluetooth adapter is acceptable. `l2_psm` specifies the PSM channel to connect to / listen on, and is a multibyte value requiring the correct byte ordering.

The `l2cap_options` structure is used in conjunction with the `getsockopt()` and `setsockopt()` functions to adjust the L2CAP maximum transmission unit (MTU).

```
struct l2cap_options {
      __u16 omtu;
      __u16 imtu;
      __u16 flush_to;
      __u8        mode;
};
```

The `omtu` field specifies the outgoing MTU whilst the `imtu` specifies the incoming MTU. The default MTU is 672 bytes, with a minimum of 48 bytes and a maximum of 65.535 bytes allowed. The remaining fields of the `l2cap_options` structure are reserved for future use.

The `hci_conn_info` structure is used in `ioctl()` request (like `HCIGETCONNINFO`) to obtain information on a specific Bluetooth connection.

```
struct hci_conn_info {
      __u16 handle;
      bdaddr_t    bdaddr;
      __u8        type;
      __u8        out;
      __u16 state;
      __u32 link_mode;
};
```

The `hci_dev_info` structure is used in `ioctl()` request (like `HCIGETDEVINFO`) to obtain information on the current Bluetooth device hardware and radio link parameters.

```
struct hci_dev_info {
      __u16 dev_id;
      char        name[8];
      bdaddr_t    bdaddr;
      __u32 flags;
      __u8        type;
      __u8        features[8];
      __u32 pkt_type;
      __u32 link_policy;
      __u32 link_mode;
      __u16 acl_mtu;
      __u16 acl_pkts;
      __u16 sco_mtu;
      __u16 sco_pkts;
      struct      hci_dev_stats stat;
};
```

The `hci_dev_stats` structure is included in the abovementioned `hci_dev_info` structure and is used to communicate the statistics counters of the Bluetooth device.

```
struct hci_dev_stats {
      __u32 err_rx;
      __u32 err_tx;
      __u32 cmd_tx;
      __u32 evt_rx;
      __u32 acl_tx;
      __u32 acl_rx;
      __u32 sco_tx;
      __u32 sco_rx;
      __u32 byte_rx;
      __u32 byte_tx;
};
```

The `sdp_list_t` structure provides a linked list implementation to organise SDP search results:

```
typedef struct _sdp_list_t {
    struct _sdp_list_t *next;
    void *data;
} sdp_list_t;
```

Use the `sdp_list_append()` and `sdp_list_free()` functions to grow or kill the list.

The `sdp_record_t` represents a single service record being advertised by another device:

```
typedef struct {
      uint32_t handle;
      sdp_list_t *pattern;
      sdp_list_t *attrlist;
} sdp_record_t;
```

B.2 Common Bluetooth functions

Following is a short description of commonly used Bluetooth function calls for interfacing with the Linux Bluetooth subsystem:

```
int hci_get_route( bdaddr_t *bdaddr );
int hci_devid( "01:23:45:67:89:AB" );
```

These functions will both obtain and return the `dev_id` (see the previous section) resource number of the Bluetooth device by using the parameters supplied. After obtaining the `dev_id`, a socket connection to the Bluetooth host controller can be opened via the following function call:

```
int hci_open_dev( int dev_id );
```

After the device is open, `ioctl()` calls can be used to communicate with it. A number of `hci_()` utility functions is also available, as discussed below.

The `hci_inquiry()` function is used to discover nearby Bluetooth devices and completes an `inquiry_info` structure described in the previous section:

```
int hci_inquiry(int dev_id, int len, int max_rsp, const uint8_t *lap,
inquiry_info **ii, long flags);
```

The `hci_read_remote_name()` function is used to determine the user-friendly name associated with the Bluetooth address of a remote device.

```
int hci_read_remote_name(int sock, const bdaddr_t *ba, int len, char
*name, int timeout)
```

Sockets to open RFCOMM Bluetooth connections are allocated in a similar way than with traditional Internet programming:

```
s = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);
```

L2CAP provides reliable datagram-oriented connections with packets delivered in order, so the `SOCK_SEQPACKET` socket type can be used. For opening an L2CAP socket the function call will look like:

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

Other TCP-like networking functions like `bind()`, `connect()`, `listen()`, `accept()` and `close()` are also used, with `read()`, `write()`, `send()` and `recv()` used for data transfers.

Network data transmission however uses big-endian byte ordering, whereas Bluetooth make use of little-endian byte ordering. The functions `htobs()`, `btohs()`, `htobl()` and `btohl()` can be used to convert between byte orderings.

The `getsockopt()` and `setsockopt()` functions is used in conjunction with the `l2cap_options` structure, described in the previous section, to adjust the L2CAP maximum transmission unit (MTU):

```
static int getsockopt(struct socket *sock,
             int lvl, int opt, char __user *ov, int __user *ol)

static int setsockopt(struct socket *sock,
             int lvl, int opt, char __user *ov, int ol)
```

For more precise control over a connection or obtaining low level device information, the `hci_send_cmd()` function can be used to send commands to the microcontroller on the local Bluetooth device. A command consists of an Opcode Group Field (OGF) that specifies the general command category, and an Opcode Command Field (OCF) that specifies the actual command, followed by a series of command parameters.

```
int hci_send_cmd(int sock, uint16_t ogf, uint16_t ocf,
            uint8_t plen, void *param);
```

Calling `read()` on an open HCI socket waits for and receives the next event form the microcontroller, consisting of a header field specifying the event type, and the event parameters.

`sdp_connect()` is used to connect to the local SDP server followed by a call to `sdp_record_register()` to register a new advertised service.

B.3 Sample Bluetooth user program

This section list the C source code of a Linux Bluetooth application called `btest` that implements the most common BlueZ API calls [31] for interfacing to the Linux Bluetooth subsystem from user space. The application also implements common Bluetooth socket functions grouped together in the `btapi` module listed in Addendum B.4.

```
/***************************************************************
* Bluetooth Test User Level Application
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
****************************************************************
*
* This user level application was developed partly for investigating
* Bluetooth programming techniques under Linux and partly for
* investigating Bluetooth data flow through the Linux subsystem.
*
* The program makes use of a set of functions grouped together
* in btapi.
*
* The command line driven application can setup server and client
* connections over Bluetooth's RFCOMM and L2CAP interfaces using sockets,
* and make use of the Service Discovery Protocol to probe the SDP entries
* of a remote device. The program usage is a follows:
*
* btest OPERATION [BDADDRESS] [CHANNEL/PSM NUMBER/TIMEOUT] [DATA/SIZE]
*
*       Operations:
*               rs      Start an RFCOMM server connection on channel [CHANNEL NUMBER]
*
*               rc      Start an RFCOMM client connection to [BDADDRESS] on channel [CHANNEL NUMBER]
*
*               ls      Start an L2CAP server connection on PSM [PSM NUMBER]
*
*               lc      Start an L2CAP client connection to [BDADDRESS] on PSM [PSM NUMBER]
*
*               s       Scan for nearby Bluetooth Devices
*
*               t       Set Timeout of [BDADDRESS] to [TIMEOUT]
*
*               ds      SDP scan device [BDADDRESS] for advertised service [UUID]
*
*               dr      Register service [UUID] with local SDP server
*
***************************************************************/
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <ctype.h>
#include "../btapi/btapi.h"

//#define BTEST_DEBUG 1
/***********************************************************************************************
* Debug functions
***********************************************************************************************/
/* Macros to help debugging */
#undef PDEBUG /* undef it, just in case */
#ifdef BTEST_DEBUG
#define PDEBUG(fmt, args...) printf("btest> " fmt, ## args)
#else
#define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

/***********************************************************************************************
* Bluetooth RFCOMM functions
***********************************************************************************************/

/*************************************************
* btest_rfcomm_server()
*
* Function for managing a server connection over RFCOMM.
*
* uint8_t channel - RFCOMM channel to use
*
* return - 0 on success, negative on failure
*************************************************/
int btest_rfcomm_server(uint8_t channel)
{
        int dev_id;
```

```
    struct sockaddr_rc rem_addr = { 0 };
    char buf[1024] = { 0 };
    int server, client, bytes_read;
    socklen_t opt = sizeof(rem_addr);

    // open a server socket and wait for a single connection
    server = btapi_rfcomm_open_server_socket(channel);
    if(server <= 0)
    {
        printf("Error opening server socket! \n");
        return -1;
    }

    // accept one connection
    client = accept(server, (struct sockaddr *)&rem_addr, &opt);
    if(client <= 0)
    {
        printf("Error accepting client on socket! \n");
        return -2;
    }
    ba2str( &rem_addr.rc_bdaddr, buf );
    fprintf(stderr, "Accepted connection from %s\n", buf);

    // clear the buffer
    memset(buf, 0, sizeof(buf));

    // read data from the client
    bytes_read = read(client, buf, sizeof(buf));
    if( bytes_read > 0 ) {
        printf("Received: %s\n", buf);
    }
    else
    {
        printf("Error reading on socket %d! \n", client);
        return -3;
    }

    // close connection
    close(client);
    // close the parent server socket
    close(server);
    return 0;
} « end btest_rfcomm_server »

/**************************************************
* btest_rfcomm_client()
*
* Function for opening a client connection over RFCOMM to a
* remote Bluetooth device.
*
* bdaddr_t bdaddr - BT address of target device
* uint8_t channel - RFCOMM channel to use
*
* return - 0 on success, negative on failure
**************************************************/
int btest_rfcomm_client(bdaddr_t bdaddr, uint8_t channel)
{
    int bt_sock, status;

    bt_sock = btapi_rfcomm_open_client_socket(bdaddr, channel);
    if(bt_sock <= 0)
    {
        printf("Error opening socket! \n");
        return -1;
    }

    // send a message
    status = write(bt_sock, "hello! ", 6);
    if(status < 0)
    {
        printf("Error writing on socket %d! \n", bt_sock);
        return -2;
    }
    close(bt_sock);
    return 0;
} « end btest_rfcomm_client »

/**************************************************************************************************
* Bluetooth L2CAP functions
**************************************************************************************************/

/**************************************************
* btest_l2cap_server()
*
* Function for opening a server connection over L2CAP
*
* uint16_t psm - PSM channel to use
*
* return - 0 on success, negative on failure
**************************************************/
int btest_l2cap_server(uint16_t psm)
{
    struct sockaddr_l2 loc_addr = { 0 }, rem_addr = { 0 };
    char buf[1024] = { 0 };
    int server, client, bytes_read;
    socklen_t opt = sizeof(rem_addr);
    int i;

    // open a BT L2CAP server socket
    server = btapi_l2cap_open_server_socket(psm);
    if(server <= 0)
```

```
        {
                printf("Error opening server socket! \n");
                return - 1;
        }

        / / accept one connection
        PDEBUG("Accepting L2CAP socket: accept()...\n");
        client = accept(server, (struct sockaddr *)&rem_addr, &opt);
        if(client <= 0)
        {
                printf("Error accepting client on socket! \n");
                return - 2;
        }
        PDEBUG("Accepted client socket %d on server %d: accept()...Done! \n", client, server);
        ba2str( &rem_addr.l2_bdaddr, buf );
        fprintf(stderr, "accepted connection from %s\n", buf);
        memset(buf, 0, sizeof(buf));

        / / read data from the client
        for(i=0;i<4;i++)
        {
                PDEBUG("\n\nReading L2CAP client socket %d pass %d: read()...\n", client, i);
                bytes_read = read(client, buf, sizeof(buf));
                PDEBUG("Read %d on socket: read()...Done! \n", bytes_read);
                if( bytes_read > 0 ) {
                        printf("received [%s]\n", buf);
                        PDEBUG("Received [%s]\n", buf);
                }
                else
                {
                        printf("Error reading on socket %d\n", client);
                }
        }

        / / close connection
        PDEBUG("\n\nClosing client socket %d: close()...\n", client);
        close(client);
        PDEBUG("Closing client socket: close()...Done! \n");
        PDEBUG("Closing client socket %d: close()...\n", server);
        close(server);
        PDEBUG("Closing server socket: close()...Done! \n");
        return 0;
} « end btest_l2cap_server »

/ **************************************************
* btest_l2cap_client()
*
* Function for opening a client connection over L2CAP to a
* remote Bluetooth device.
*
* bdaddr_t bdaddr - BT address of target device
* uint16_t psm - PSM channel to use
*
* return - 0 on success, negative on failure
**************************************************/
int btest_l2cap_client(bdaddr_t bdaddr, uint16_t psm)
{
        int bt_sock, status;
        char message[8] = "M0: 202\0";
        int i;

        bt_sock = btapi_l2cap_open_client_socket(bdaddr, psm);
        if(bt_sock <= 0)
        {
                printf("Error opening socket! \n");
                return - 1;
        }

        / / send a message
        PDEBUG("Connected...\n\n\n");
        for(i=0;i<4;i++)
        {
                message[1] = (char) ('0' + (char)i);
                PDEBUG("Writing message %s on socket %d: write()...\n", message, bt_sock);
                status = write(bt_sock, message, 8);
                if( status < 0 )
                {
                        printf("Error writing on socket %d! \n", bt_sock);
                        return - 2;
                }
                PDEBUG("Writing message: write()...Done! \n\n\n");
                getc(stdin);
        }

        PDEBUG("Closing socket %d: close()...\n", bt_sock);
        close(bt_sock);
        PDEBUG("Closing socket: close()...Done! \n");
        return 0;
} « end btest_l2cap_client »

/ ***************************************************************************************************
* Application Commandline Usage
***************************************************************************************************/

/ **************************************************
* print_usage()
*
* Displays information on how to use this program
*
* return - void
**************************************************/
```

```
void print_usage(void)
{
        printf("\nUsage: btest OPERATION [BDADDRESS] [CHANNEL/ PSM NUMBER/ TIMEOUT] [DATA/ SIZE]\n\n");
        printf("Operations:\n");
        printf("\trs\tStart an RFCOMM server connection on channel [CHANNEL NUMBER]\n\n");
        printf("\trc\tStart an RFCOMM client connection to [BDADDRESS] on channel [CHANNEL NUMBER]\n\n");
        printf("\tls\tStart an L2CAP server connection on PSM [PSM NUMBER]\n\n");
        printf("\tlc\tStart an L2CAP client connection to [BDADDRESS] on PSM [PSM NUMBER]\n\n");
        printf("\ts\tScan for nearby Bluetooth Devices\n\n");
        printf("\tt\tSet Timeout of [BDADDRESS] to [TIMEOUT]\n\n");
        printf("\tds\tSDP scan device [BDADDRESS] for advertised service [UUID]\n\n");
        printf("\tdr\tRegister service [UUID] with local SDP server\n\n");
}

/ **************************************************
* simple_strtoul()
*
* Conversion of commandline hex values in string format to unsigned long
*
* const char *cp - pointer to string to be converted
* char **endp - pointer to end of string
* unsigned int base - 16 for hex, 10 for natural
*
* return - the converted value
**************************************************/
unsigned long simple_strtoul(const char *cp,char **endp,unsigned int base)
{
        unsigned long result = 0,value;

        if (! base) {
                base = 10;
                if (*cp == '0') {
                        base = 8;
                        cp++;
                        if ((toupper(*cp) == 'X') && isxdigit(cp[1])) {
                                cp++;
                                base = 16;
                        }
                }
        } else if (base == 16) {
                if (cp[0] == '0' && toupper(cp[1]) == 'X')
                cp += 2;
        }
        while (isxdigit(*cp) &&
        (value = isdigit(*cp) ? *cp- '0' : toupper(*cp)- 'A'+10) < base) {
                result = result*base + value;
                cp++;
        }
        if (endp)
                *endp = (char *)cp;
        return result;
} « end simple_strtoul »

/ **************************************************
* main()
*
* Main Entry point
* Parse the command line input to determine which
* Bluetooth connection type to setup.
*
* int argc - number of commandline argurments
* char* argv - the commandline argurments
*
* return - 0 on success
**************************************************/
int main(int argc, char* argv[])
{
        char operation = 's';
        char server_client = 's';
        bdaddr_t bdaddr;
        uint8_t channel = 1;
        uint16_t psm = 4099;
        int timeout = 0;
        uint16_t uuid = 0;
        char bt_info[200];

        if(argc < 2)
        {
                print_usage();
                return 1;
        }
        operation = toupper(argv[1][0]);
        switch(operation)
        {
                case 'S':
                btapi_inquire(bt_info);
                printf("%s\n", bt_info);
                break;
                case 'T':
                if((argc > 2) && (strlen(argv[2]) == 17))
                {
                        str2ba(argv[2], &bdaddr);
                        if(argc > 3) timeout = (uint8_t)simple_strtoul(argv[3],NULL,0);
                        btapi_set_flush_timeout(&bdaddr, timeout);
                } else print_usage();
                break;

                case 'D':
                if(toupper(argv[1][1]) == 'R')
```

```
            {
                if(argc > 2) uuid = (uint16_t)simple_strtoul(argv[2],NULL,0);
                btapi_sdp_register_service(uuid);
            }
            else
            {
                if((argc > 2) && (strlen(argv[2]) == 17))
                {
                    str2ba(argv[2], &bdaddr);
                    if(argc > 3) uuid = (uint16_t)simple_strtoul(argv[3],NULL,0);
                    btapi_sdp_query(bdaddr, uuid);
                } else print_usage();
            }
            break;

            case 'R':
            server_client = toupper(argv[1][1]);
            if(server_client == 'C') / * Client */
            {
                if((argc > 2) && (strlen(argv[2]) == 17))
                {
                    str2ba(argv[2], &bdaddr);
                    if(argc > 3) channel = (uint8_t)simple_strtoul(argv[3],NULL,0);
                    btest_rfcomm_client(bdaddr, channel);
                } else print_usage();
            } else if (server_client == 'S') / * Server */
            {
                if(argc > 2) channel = (uint8_t)simple_strtoul(argv[2],NULL,0);
                btest_rfcomm_server(channel);
            } else print_usage();
            break;

            case 'L':
            server_client = toupper(argv[1][1]);
            if(server_client == 'C') / * Client */
            {
                if((argc > 2) && (strlen(argv[2]) == 17))
                {
                str2ba(argv[2], &bdaddr);
                if(argc > 3) psm = (uint16_t)simple_strtoul(argv[3],NULL,0);
                btest_l2cap_client(bdaddr, psm);
                } else print_usage();
            } else if (server_client == 'S') / * Server */
            {
                if(argc > 2) psm = (uint16_t)simple_strtoul(argv[2],NULL,0);
                btest_l2cap_server(psm);
            } else print_usage();
            break;
            default:
            print_usage();
        } « end switch operation »

        return 0;
} « end main »
```

B.4 Bluetooth common socket interface module

The user level functions for interfacing with the Bluetooth socket system [31] were grouped together in a single reusable module called `btapi` for easy implementation in different user applications. The source of this module is listed here:

B.4.1 Btapi Header File

```
#ifndef _BTAPI_H_
#define _BTAPI_H_

#include <sys/ socket.h>
#include <bluetooth/ bluetooth.h>
#include <bluetooth/ sdp.h>
#include <bluetooth/ sdp_lib.h>
#include <bluetooth/ rfcomm.h>
#include <bluetooth/ l2cap.h>
#include <bluetooth/ sco.h>

int btapi_inquire(char *target_string);
int btapi_set_flush_timeout(bdaddr_t *ba, int timeout);
int btapi_rfcomm_open_server_socket(uint8_t channel);
int btapi_rfcomm_open_client_socket(bdaddr_t bdaddr, uint8_t channel);
int btapi_l2cap_check_psm(uint16_t psm);
int btapi_l2cap_set_mtu( int sock, uint16_t mtu );
int btapi_l2cap_open_server_socket(uint16_t psm);
int btapi_l2cap_open_client_socket(bdaddr_t bdaddr, uint16_t psm);
int btapi_sco_open_server_socket(void);
int btapi_sco_open_client_socket(bdaddr_t bdaddr);
int btapi_hci_open_server_socket(void);
int btapi_hci_open_client_socket(bdaddr_t bdaddr);
void btapi_uuid2str(uint8_t arr_uuid[16], char* str_uuid);
int btapi_sdp_query(bdaddr_t target, uint16_t ask_uuid);
sdp_session_t *btapi_sdp_register_service(uint16_t ask_uuid);

#endif / * _BTAPI_H_ */
```

B.4.2 Btapi Source File

```
/ **************************************************************
* Bluetooth Test User Level Application
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/ or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
**************************************************************
*
* This user level application was developed partly for investigating
* Bluetooth programming techniques under Linux and partly for
* investigating Bluetooth data flow through the Linux subsystem.
*
* The command line driven application can setup server and client
* connection over Bluetooth's RFCOMM and L2CAP interfaces using sockets,
* and make use of the Service Discovery Protocol to probe the SDP entries
* of a remote device. The program usage is a follows:
*
**************************************************************/

#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ ioctl.h>
#include <ctype.h>
#include <sys/ socket.h>
#include <bluetooth/ bluetooth.h>
#include <bluetooth/ sdp.h>
#include <bluetooth/ sdp_lib.h>
#include <bluetooth/ rfcomm.h>
#include <bluetooth/ l2cap.h>
#include <bluetooth/ sco.h>
#include <bluetooth/ hci.h>
#include <bluetooth/ hci_lib.h>
```

```
#include <errno.h>
#include "btapi.h"

//#define BTAPI_DEBUG 1
#define LISTEN_LENGTH 1

/*****************************************************************************************************
* Debug functions
*****************************************************************************************************/

/* Macros to help debugging */
#undef PDEBUG /* undef it, just in case */
#ifdef BTAPI_DEBUG
#define PDEBUG(fmt, args...) printf("btapi> " fmt, ## args)
#else
#define PDEBUG(fmt, args...) /* not debugging: nothing */
#define WAIT_IN
#endif

/*****************************************************************************************************
* General Bluetooth Functions
*****************************************************************************************************/

/**************************************************
* find_conn()
*
* Search local device for a connection to a remote Bluetooth address
*
* int dev_id - Local HCI device ID
* bdaddr_t *bdaddr - Remote Bluetooth Address
*
* return - always 0
**************************************************/
uint16_t find_conn(int dev_id, bdaddr_t *bdaddr)
{
	struct hci_conn_list_req *cl;
	struct hci_conn_info *ci;
	int i, s;
	uint16_t rv = (uint16_t)-1;

	s = socket(AF_BLUETOOTH, SOCK_RAW, BTPROTO_HCI);
	if (-1 == s) {
		return (uint16_t)-1;
	}
	if (!(cl = malloc(10 * sizeof(*ci) + sizeof(*cl)))) {
		close(s);
		return (uint16_t)-1;
	}
	cl->dev_id = dev_id;
	cl->conn_num = 10;
	ci = cl->conn_info;
	if (ioctl(s, HCIGETCONNLIST, (void*)cl)) {
		free(cl);
		close(s);
		return (uint16_t)-1;
	}
	for (i=0; i < cl->conn_num; i++, ci++) {
		if (!bacmp(bdaddr, &ci->bdaddr) && ci->type == ACL_LINK) {
			rv=ci->handle;
			break;
		}
	}

	free(cl);
	close(s);
	return rv;
} « end find_conn »

/**************************************************
* btapi_hci_link_info()
*
* Obtain link information on a connection to a remote Bluetooth device
*
* bdaddr_t bdaddr - Bluetooth address of remote connection
* unsigned char *lq - Link quality value to remote connection
* unsigned char *rssi - Link RSSI of remote connection
* unsigned char *txpwr - Transmit Power of remote connection
*
* return - always 0
**************************************************/
int btapi_hci_link_info(bdaddr_t bdaddr, unsigned char *lq, unsigned char *rssi, unsigned char *txpwr)
{
	int dd, rv;
	uint16_t handle;
	int dev_id;
	int status;

	dev_id = hci_get_route(NULL);
	dd = hci_open_dev(dev_id);

	/* get a connection */
	handle = find_conn(dev_id, &bdaddr);
	status = hci_read_link_quality(dd, handle, lq, 100);
	status = hci_read_rssi(dd, handle, rssi, 100);
	status = hci_read_transmit_power_level(dd, handle, 0, txpwr, 100);
	PDEBUG("LQ is %d, RSSI is %d, TXPWR is %d\n", *lq, *rssi, *txpwr);

	return status;
} « end btapi_hci_link_info »

/**************************************************
```

```
* btapi_inquire()
*
* Scan for remote Bluetooth devices and display its information
*
* char *target_string - Target string that should get the
* obtained information
*
* return - always 0
*************************************************/
int btapi_inquire(char *target_string)
{
	inquiry_info *ii = NULL;
	int max_rsp, num_rsp;
	int dev_id, sock, len, flags;
	int i;
	char addr[19] = { 0 };
	char name[248] = { 0 };

	dev_id = hci_get_route(NULL);
	/ * Note - socket represents connection to local Bluetooth controller and not remote BT device */
	sock = hci_open_dev( dev_id );
	if (dev_id < 0 || sock < 0) {
		perror("opening socket");
		exit(1);
	}

	len = 8;
	max_rsp = 255;
	flags = IREQ_CACHE_FLUSH;
	ii = (inquiry_info*)malloc(max_rsp * sizeof(inquiry_info));

	memset(target_string, 0, sizeof(target_string));

	num_rsp = hci_inquiry(dev_id, len, max_rsp, NULL, &ii, flags);
	if( num_rsp < 0 ) perror("hci_inquiry");

	for (i = 0; i < num_rsp; i++) {
		ba2str(&(ii+i)->bdaddr, addr);
		memset(name, 0, sizeof(name));
		if (hci_read_remote_name(sock, &(ii+i)->bdaddr, sizeof(name), name, 0) < 0)
			strcpy(name, "[unknown]");
		sprintf(target_string, "%s %s\n", addr, name);
	}

	free( ii );
	close( sock );
	return 0;
} « end btapi_inquire »

/ *************************************************
* set_flush_timeout()
*
* Change the packet timeout for a connection to <timeout> x 0.625 ms
*
* bdaddr_t *ba - address of target Bluetooth device
* int timeout - the new timeout value
*
* return - error code or responce of HCI send command
*************************************************/
/ * Change the packet timeout for a connection to <timeout> x 0.625 ms */
int btapi_set_flush_timeout(bdaddr_t *ba, int timeout)
{
	int err = 0, dd;
	struct hci_conn_info_req *cr = 0;
	struct hci_request rq = { 0 };
	struct
	{
	uint16_t handle;
	uint16_t flush_timeout;
	} cmd_param;
	struct
	{
	uint8_t status;
	uint16_t handle;
	} cmd_response;

	/ / find the connection handle to the specified bluetooth device
	cr = (struct hci_conn_info_req*) malloc(sizeof(struct hci_conn_info_req) + sizeof(struct hci_conn_info));
	bacpy( &cr->bdaddr, ba );
	cr->type = ACL_LINK;
	dd = hci_open_dev( hci_get_route( &cr->bdaddr ) );
	if( dd < 0 )
	{
		err = dd;
		goto cleanup;
	}
	err = ioctl(dd, HCIGETCONNINFO, (unsigned long) cr );
	if( err ) goto cleanup;

	/ / build a command packet to send to the bluetooth microcontroller
	cmd_param.handle = cr->conn_info->handle;
	cmd_param.flush_timeout = htobs(timeout);
	rq.ogf = OGF_HOST_CTL;
	rq.ocf = 0x28;
	rq.cparam = &cmd_param;
	rq.clen = sizeof(cmd_param);
	rq.rparam = &cmd_response;
	rq.rlen = sizeof(cmd_response);
	rq.event = EVT_CMD_COMPLETE;
```

```
        / / send the command and wait for the response
        err = hci_send_req( dd, &rq, 0 );
        if( err ) goto cleanup;
        if( cmd_response.status )
        {
                err = - 1;
                errno = bt_error(cmd_response.status);
        }

        cleanup:
        free(cr);
        if( dd >= 0) close(dd);
        return err;
} « end btapi_set_flush_timeout »

/ ****************************************************************************************************
* Bluetooth RFCOMM functions
****************************************************************************************************/

/ **************************************************
* btapi_rfcomm_open_server_socket()
*
* Function for opening a server connection over RFCOMM.
*
* uint8_t channel - RFCOMM channel to use
*
* return - integer handle to the socket on success, negative on failure
**************************************************/
int btapi_rfcomm_open_server_socket(uint8_t channel)
{
        int dev_id;
        struct sockaddr_rc loc_addr = { 0 };
        int server;

        dev_id = hci_get_route(NULL);
        if((channel < 1) | (channel > 30))
        {
                printf("Error: Channel number is out of the range 0 - 30! \n");
                return - 2;
        }
        printf("\nOpening RFCOMM server on HCI%d with channel %d...\n\n", dev_id, channel);

        / / allocate socket
        server = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

        / / bind socket to port 1 of the first available local bluetooth adapter
        loc_addr.rc_family = AF_BLUETOOTH;
        loc_addr.rc_bdaddr = *BDADDR_ANY;
        loc_addr.rc_channel = (uint8_t) channel;
        if (bind(server, (struct sockaddr *)&loc_addr, sizeof(loc_addr)) < 0)
        {
                printf("Can't bind RFCOMM socket: Address already in use! \n");
                close(server);
                return - 1;
        }
        / / put socket into listening mode
        listen(server, LISTEN_LENGTH);
        return server;
} « end btapi_rfcomm_open_server_socket »

/ **************************************************
* btapi_rfcomm_open_client_socket()
*
* Function for opening a client connection over RFCOMM to a
* remote Bluetooth device.
*
* bdaddr_t bdaddr - BT address of target device
* uint8_t channel - RFCOMM channel to use
*
* return - integer handle to the socket on success, negative on failure
**************************************************/
int btapi_rfcomm_open_client_socket(bdaddr_t bdaddr, uint8_t channel)
{
        int dev_id;
        char addr_string[19] = { 0 };
        struct sockaddr_rc addr = { 0 };
        int client, status;

        if((channel < 1) | (channel > 30))
        {
                printf("Error: Channel number is out of the range 0 - 30! \n");
                return - 2;
        }

        dev_id = hci_get_route(NULL);
        ba2str(&bdaddr, addr_string);
        printf("\nOpening RFCOMM client to %s on HCI%d with channel %d...\n\n", addr_string, dev_id, channel);

        / / allocate a socket
        client = socket(AF_BLUETOOTH, SOCK_STREAM, BTPROTO_RFCOMM);

        / / set the connection parameters (who to connect to)
        addr.rc_family = AF_BLUETOOTH;
        addr.rc_channel = (uint8_t) channel;
        addr.rc_bdaddr = bdaddr;

        / / connect to server
        status = connect(client, (struct sockaddr *)&addr, sizeof(addr));
        if(status ! = 0) return status;

        return client;
```

```
} « end btapi_rfcomm_open_client_socket »

/ ***********************************************************************************************
* Bluetooth L2CAP functions
***********************************************************************************************/

/ *************************************************
* btapi_l2cap_check_psm()
*
* Check if the provided PSM is a valid one
*
* uint16_t psm - psm to check
*
* return - 1 if valid, negative if not
*************************************************/
int btapi_l2cap_check_psm(uint16_t psm)
{
        if((psm < 4097) || (psm > 32765)) return - 1;
        if((psm% 2) == 0) return - 2;
        return 1;
}

/ *************************************************
* btapi_l2cap_set_mtu()
*
* Sets the maximum transmission unit for an L2CAP connection
*
* int sock - the socket to use
* uint16_t mtu - the new mtu value
*
* return - error code
*************************************************/
int btapi_l2cap_set_mtu( int sock, uint16_t mtu )
{
        struct l2cap_options opts;
        int optlen = sizeof(opts), err;

        err = getsockopt( sock, SOL_L2CAP, L2CAP_OPTIONS, &opts, &optlen );
        if( ! err )
        {
                opts.omtu = opts.imtu = mtu;
                err = setsockopt( sock, SOL_L2CAP, L2CAP_OPTIONS, &opts, optlen );
        }
        return err;
};

/ *************************************************
* btapi_l2cap_get_mtu()
*
* Get the maximum transmission unit for an L2CAP connection
*
* int sock - the socket to use
*
* return - uint16_t mtu
*************************************************/
uint16_t btapi_l2cap_get_mtu( int sock)
{
        struct l2cap_options opts;
        int optlen = sizeof(opts), err;

        err = getsockopt( sock, SOL_L2CAP, L2CAP_OPTIONS, &opts, &optlen );
        if( ! err )
        {
                PDEBUG("Obtained L2CAP MTU of %d\n", opts.omtu);
                return opts.omtu;
        }
        return err;
};

/ *************************************************
* btapi_l2cap_open_server_socket()
*
* Function for opening a server connection over L2CAP
*
* uint16_t psm - PSM channel to use
*
* return - integer handle to the socket on success, negative on failure
*************************************************/
int btapi_l2cap_open_server_socket(uint16_t psm)
{
        int dev_id;
        struct sockaddr_l2 loc_addr = { 0 };
        int server;

        if(btapi_l2cap_check_psm(psm) < 0)
        {
                printf("Error: PSM number is not odd or out of range 4097 - 32765! \n");
                return - 2;
        }
        PDEBUG("Obtaining hci route: hci_get_route()...\n");
        dev_id = hci_get_route(NULL);
        PDEBUG("Obtained dev_id %d: hci_get_route()...Done! \n", dev_id);
        printf("\nOpening L2CAP server on HCI%d with PSM %d...\n\n", dev_id, psm);

        / / allocate socket
        PDEBUG("Allocating L2CAP socket: socket()...\n");
        server = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
        PDEBUG("Allocating L2CAP socket %d: Done! \n", server);

        / / bind socket to port 0x1001 of the first available
        / / bluetooth adapter
```

```
	loc_addr.l2_family = AF_BLUETOOTH;
	loc_addr.l2_bdaddr = *BDADDR_ANY;
	loc_addr.l2_psm = htobs(psm);
	PDEBUG("Binding L2CAP socket %d on PSM %d: bind()...\n", server, psm);
	bind(server, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
	PDEBUG("Binding L2CAP socket: bind()...Done! \n");

	/ / put socket into listening mode
	PDEBUG("Listening on L2CAP socket %d: listen()...\n", server);
	listen(server, LISTEN_LENGTH);
	PDEBUG("Listening on L2CAP socket: listen()...Done! \n");
	return server;
} « end btapi_l2cap_open_server_socket »

/ ***************************************************
* btapi_l2cap_open_client_socket()
*
* Function for opening a client connection over L2CAP to a
* remote Bluetooth device.
*
* bdaddr_t bdaddr - BT address of target device
* uint16_t psm - PSM channel to use
*
* return - integer handle to the socket on success, negative on failure
****************************************************/
int btapi_l2cap_open_client_socket(bdaddr_t bdaddr, uint16_t psm)
{
	int dev_id;
	char addr_string[19] = { 0 };
	struct sockaddr_l2 addr = { 0 };
	int client, status;

	PDEBUG("Obtaining hci route: hci_get_route()...\n");
	dev_id = hci_get_route(NULL);
	PDEBUG("Obtained dev_id %d: hci_get_route()...Done! \n", dev_id);
	if(btapi_l2cap_check_psm(psm) < 0)
	{
		printf("Error: PSM number is not odd or out of range 4097 - 32765! \n");
		return - 2;
	}
	ba2str(&bdaddr, addr_string);
	printf("\nOpening L2CAP client to %s on HCI%d with PSM %d...\n\n", addr_string, dev_id, psm);

	/ / allocate a socket
	PDEBUG("Allocating L2CAP socket: socket()...\n");
	client = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_L2CAP);
	PDEBUG("Allocating L2CAP socket %d: Done! \n", client);

	/ / set the connection parameters (who to connect to)
	addr.l2_family = AF_BLUETOOTH;
	addr.l2_psm = htobs(psm);
	addr.l2_bdaddr = bdaddr;

	/ / connect to client
	PDEBUG("Connecting on L2CAP socket %d with PSM %d: socket()...\n", client, psm);
	status = connect(client, (struct sockaddr *)&addr, sizeof(addr));
	if(status ! = 0) return status;
	PDEBUG("Done connecting: socket()...Done! \n");
	return client;
} « end btapi_l2cap_open_client_socket »

/ ***********************************************************************************************
* Bluetooth SCO functions
***********************************************************************************************/

/ ***************************************************
* btapi_sco_set_mtu()
*
* Sets the maximum transmission unit for an SCO connection
*
* int sock - the socket to use
* uint16_t mtu - the new mtu value
*
* return - error code
****************************************************/
int btapi_sco_set_mtu( int sock, uint16_t mtu )
{
	struct sco_options opts;
	int optlen = sizeof(opts), err;

	err = getsockopt( sock, SOL_SCO, SCO_OPTIONS, &opts, &optlen );
	if( ! err )
	{
		opts.mtu = mtu;
		err = setsockopt( sock, SOL_SCO, SCO_OPTIONS, &opts, optlen );
	}
	return err;
};

/ ***************************************************
* btapi_sco_get_mtu()
*
* Get the maximum transmission unit for an SCO connection
*
* int sock - the socket to use
*
* return - uint16_t mtu
****************************************************/
uint16_t btapi_sco_get_mtu( int sock)
{
	struct sco_options opts;
```

```
        int optlen = sizeof(opts), err;

        err = getsockopt( sock, SOL_SCO, SCO_OPTIONS, &opts, &optlen );
        if( ! err )
        {
                PDEBUG("Obtained SCO MTU of %d\n", opts.mtu);
                return opts.mtu;
        }
        return err;
};

/ *************************************************
* btapi_sco_open_server_socket()
*
* Function for opening a server connection over SCO
*
*
* return - integer handle to the socket on success, negative on failure
*************************************************/
int btapi_sco_open_server_socket(void)
{
        int dev_id;
        struct sockaddr_sco loc_addr = { 0 };
        int server;

        PDEBUG("Obtaining hci route: hci_get_route()...\n");
        dev_id = hci_get_route(NULL);
        PDEBUG("Obtained dev_id %d: hci_get_route()...Done! \n", dev_id);
        printf("\nOpening SCO server on HCI%d\n\n", dev_id);

        / / allocate socket
        PDEBUG("Allocating SCO socket: socket()...\n");
        server = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_SCO);
        PDEBUG("Allocating SCO socket %d: Done! \n", server);

        / / bind socket to port 0x1001 of the first available
        / / bluetooth adapter
        loc_addr.sco_family = AF_BLUETOOTH;
        loc_addr.sco_bdaddr = *BDADDR_ANY;
        PDEBUG("Binding SCO socket %d...\n", server);
        bind(server, (struct sockaddr *)&loc_addr, sizeof(loc_addr));
        PDEBUG("Binding SCO socket: bind()...Done! \n");

        / / put socket into listening mode
        PDEBUG("Listening on SCO socket %d: listen()...\n", server);
        listen(server, LISTEN_LENGTH);
        PDEBUG("Listening on SCO socket: listen()...Done! \n");
        return server;
} « end btapi_sco_open_server_socket »

/ *************************************************
* btapi_sco_open_client_socket()
*
* Function for opening a client connection over SCO to a
* remote Bluetooth device.
*
* bdaddr_t bdaddr - BT address of target device
*
* return - integer handle to the socket on success, negative on failure
int btapi_sco_open_client_socket(bdaddr_t bdaddr)
{
        int dev_id;
        char addr_string[19] = { 0 };
        struct sockaddr_sco addr = { 0 };
        int client, status;

        PDEBUG("Obtaining hci route: hci_get_route()...\n");
        dev_id = hci_get_route(NULL);
        PDEBUG("Obtained dev_id %d: hci_get_route()...Done! \n", dev_id);
        ba2str(&bdaddr, addr_string);
        printf("\nOpening SCO client to %s on HCI%d...\n\n", addr_string, dev_id);

        / / allocate a socket
        PDEBUG("Allocating SCO socket: socket()...\n");
        client = socket(AF_BLUETOOTH, SOCK_SEQPACKET, BTPROTO_SCO);
        PDEBUG("Allocating SCO socket %d: Done! \n", client);

        / / set the connection parameters (who to connect to)
        addr.sco_family = AF_BLUETOOTH;
        addr.sco_bdaddr = bdaddr;

        / / connect to client
        PDEBUG("Connecting on SCO socket %d...\n", client);
        status = connect(client, (struct sockaddr *)&addr, sizeof(addr));
        if(status ! = 0) return status;
        PDEBUG("Done connecting: socket()...Done! \n");

        return client;
} « end btapi_sco_open_client_socket »

/ ****************************************************************************
* Bluetooth SDP functions
****************************************************************************/

/ *************************************************
* btapi_uuid2str()
*
* Helper function to convert a UUID to string format
*
* uint8_t arr_uuid[16] - UUID to convert
* char* str_uuid - the target string
*
```

```
* return - void
**************************************************/
void btapi_uuid2str(uint8_t arr_uuid[16], char* str_uuid)
{
        int i;

        for(i=0;i<15;i++)
        {
                sprintf(str_uuid + (i*3),"%02X:", arr_uuid[i]);
        }
        sprintf(str_uuid + (15*3),"%02X", arr_uuid[15]);
}

/ *************************************************
* btapi_sdp_query()
*
* Function to query a remote device's SDP records
*
* bdaddr_t target - target BT device's address
* uint16_t ask_uuid - UUID to check for
*
* return - always 0
**************************************************/
int btapi_sdp_query(bdaddr_t target, uint16_t ask_uuid)
{
        uint8_t svc_uuid_int[] = { 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0xab, 0xcd };
        uuid_t svc_uuid;
        int err;
        char buf1[1024] = { 0 };
        char buf2[1024] = { 0 };

        svc_uuid_int[14] = (uint8_t) ((ask_uuid >> 8) & 0xFF);
        svc_uuid_int[15] = (uint8_t) (ask_uuid & 0xFF);
        btapi_uuid2str(svc_uuid_int, buf1);
        ba2str( &target, buf2 );
        PDEBUG("Searching SDP records of %s for UUID %s ...\n", buf2, buf1);

        / *********************************************************/
        / * Searching devices for SDP info */
        / *********************************************************/
        sdp_list_t *response_list = NULL, *search_list, *attrid_list;
        sdp_session_t *session = 0;

        / / connect to the SDP server running on the remote machine
        session = sdp_connect( BDADDR_ANY, &target, SDP_RETRY_IF_BUSY );

        / / specify the UUID of the application we're searching for
        sdp_uuid128_create( &svc_uuid, &svc_uuid_int );
        search_list = sdp_list_append( NULL, &svc_uuid );

        / / specify that we want a list of all the matching applications' attributes
        uint32_t range = 0x0000ffff;
        attrid_list = sdp_list_append( NULL, &range );

        / / get a list of service records that have UUID 0xabcd
        err = sdp_service_search_attr_req( session, search_list, SDP_ATTR_REQ_RANGE, attrid_list, &response_list);

        / *********************************************************/
        / * Parsing the results */
        / *********************************************************/
        sdp_list_t *r = response_list;

        / / go through each of the service records
        for (; r; r = r->next )
        {
                sdp_record_t *rec = (sdp_record_t*) r->data;
                sdp_list_t *proto_list;

                / / get a list of the protocol sequences
                if( sdp_get_access_protos( rec, &proto_list ) == 0 )
                {
                        sdp_list_t *p = proto_list;

                        / / go through each protocol sequence
                        for( ; p ; p = p->next )
                        {
                                sdp_list_t *pds = (sdp_list_t*)p->data;

                                / / go through each protocol list of the protocol sequence
                                for( ; pds ; pds = pds->next )
                                {
                                        / / check the protocol attributes
                                        sdp_data_t *d = (sdp_data_t*)pds->data;
                                        int proto = 0;
                                        for( ; d; d = d->next )
                                        {
                                                switch( d->dtd )
                                                {
                                                        case SDP_UUID16:
                                                        case SDP_UUID32:
                                                        case SDP_UUID128:
                                                        proto = sdp_uuid_to_proto( &d->val.uuid );
                                                        break;
                                                        case SDP_UINT8:
                                                        if( proto == RFCOMM_UUID )
                                                        {
                                                                printf("rfcomm channel: %d\n",d->val.int8);
                                                        }
                                                        break;
                                                }
                                        }
```

```
                    } « end for ;pds;pds=pds->next »
                sdp_list_free( (sdp_list_t*)p->data, 0 );
                } « end for ;p;p=p->next »
                sdp_list_free( proto_list, 0 );
            } « end if sdp_get_access_protos... »
            printf("found service record 0x%x\n", rec->handle);
            sdp_record_free( rec );
        } « end for ;r;r=r->next »
        sdp_close(session);
        return 0;
} « end btapi_sdp_query »

/ ************************************************
* btapi_sdp_register_service()
*
* Function for registering an SDP Bluetooth service for the device
*
* uint16_t ask_uuid - uuid to register
*
* return - handle to sdp session
************************************************/
sdp_session_t *btapi_sdp_register_service(uint16_t ask_uuid)
{
        uint32_t service_uuid_int[] = { 0, 0, 0, 0xABCD };
        uint8_t rfcomm_channel = 11;
        const char *service_name = "Roto- Rooter Data Router";
        const char *service_dsc = "An experimental plumbing router";
        const char *service_prov = "Roto- Rooter";
        int err = 0;
        sdp_session_t *session = 0;
        uuid_t root_uuid, l2cap_uuid, rfcomm_uuid, svc_uuid;
        sdp_list_t *l2cap_list = 0,

        *rfcomm_list = 0,
        *root_list = 0,
        *proto_list = 0,
        *access_proto_list = 0;
        sdp_data_t *channel = 0, *psm = 0;
        sdp_record_t *record = sdp_record_alloc();
        service_uuid_int[3] = (uint32_t)ask_uuid;
        printf("Registering service %04X ...\n", ask_uuid);

        / ********************************************************/
        / * Describing a service */
        / ********************************************************/
        / / set the general service ID
        sdp_uuid128_create( &svc_uuid, &service_uuid_int );
        sdp_set_service_id( record, svc_uuid );

        / / make the service record publicly browsable
        sdp_uuid16_create(&root_uuid, PUBLIC_BROWSE_GROUP);
        root_list = sdp_list_append(0, &root_uuid);
        sdp_set_browse_groups( record, root_list );

        / / set l2cap information
        sdp_uuid16_create(&l2cap_uuid, L2CAP_UUID);
        l2cap_list = sdp_list_append( 0, &l2cap_uuid );
        proto_list = sdp_list_append( 0, l2cap_list );

        / / set rfcomm information
        sdp_uuid16_create(&rfcomm_uuid, RFCOMM_UUID);
        channel = sdp_data_alloc(SDP_UINT8, &rfcomm_channel);
        rfcomm_list = sdp_list_append( 0, &rfcomm_uuid );
        sdp_list_append( rfcomm_list, channel );
        sdp_list_append( proto_list, rfcomm_list );

        / / attach protocol information to service record
        access_proto_list = sdp_list_append( 0, proto_list );
        sdp_set_access_protos( record, access_proto_list );

        / / set the name, provider, and description
        sdp_set_info_attr(record, service_name, service_prov, service_dsc);

        / ********************************************************/
        / * Registering a service */
        / ********************************************************/
        / / connect to the local SDP server, register the service record, and disconnect
        session = sdp_connect( BDADDR_ANY, BDADDR_LOCAL, SDP_RETRY_IF_BUSY );
        err = sdp_record_register(session, record, 0);

        / / cleanup
        sdp_data_free( channel );
        sdp_list_free( l2cap_list, 0 );
        sdp_list_free( rfcomm_list, 0 );
        sdp_list_free( root_list, 0 );
        sdp_list_free( access_proto_list, 0 );
        return session;
} « end btapi_sdp_register_service »
```

B.5 Bluetooth data generator and test program source

This section list the C source code of the Linux Bluetooth analysis application called `btan` that was developed to generate different Bluetooth data for the BToUWB subsystem and control the type of this generated data. The application also implements common Bluetooth socket functions grouped together in the `btapi` module listed in Addendum B.4. See section 3.2.2.1 for a description of this application.

```
/ ***************************************************************
* Bluetooth Test User Level Application
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/ or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
***************************************************************
*
* This user level application was developed partly for investigating
* Bluetooth programming techniques under Linux and partly for
* investigating Bluetooth data flow through the Linux subsystem.
*
* The command line driven application can setup server and client
* connection over Bluetooth's RFCOMM and L2CAP interfaces using sockets,
* and make use of the Service Discovery Protocol to probe the SDP entries
* of a remote device. The program usage is a follows:
*
**************************************************************/
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ ioctl.h>
#include <ctype.h>
#include <time.h>
#include <signal.h>
#include <termios.h>

/ /#define BTAPI_DEBUG 1
#include "../ btapi/ btapi.h"

#define BTAN_DEBUG 1
#define PACKET_DEBUG 1
#define BT_STR_ADDRESS_LENGTH 17
#define DEFAULT_L2CAP_PSM 5555
#define DEFAULT_RFCOMM_CHANNEL 15
#define MAXIMUM_BUFFER_SIZE 1024
#define BTAN_SERVER_ECHO 1
#define DELAY_MULTIPLIER 100
#define EXIT_SEQUENCE "quit"
#define MAX_L2CAP_PACKET_SIZE 1000
#define MAX_SCO_PACKET_SIZE 48
#define MAX_HCI_PACKET_SIZE 672
#define MAX_RFCOMM_PACKET_SIZE 1000
#define RX_ERROR_THRESHOLD 20

/ ***********************************************************************************************
* Application Commandline Usage
***********************************************************************************************/

/ /#define CMDLINE_PARSER_DEBUG 1
#define NR_OF_CMND_LINE_OPTIONS 12
#include "cmdline_parser.h"

struct cmd_line_options_layout COMMAND_LINE_OPTIONS[NR_OF_CMND_LINE_OPTIONS] = {
{'c', "Open in 'client' mode, followed by remote address: AA:BB:CC:DD:EE:FF", 's', NULL, 0, (int)NULL},
{'t', "Traffic Type: 'l'2cap, 's'co, 'r'fcomm", 'c', NULL, 'l', (int)NULL},
{'h', "Bluetooth channel for communication", 'i', NULL, 0, (int)NULL},
{'n', "Number of packets to send", 'i', NULL, 10, (int)NULL},
{'d', "Delay between packets", 'i', NULL, 0, (int)NULL},
{'z', "Packet size", 'i', NULL, 10, (int)NULL},
{'i', "Packet size incrementation", 'i', NULL, 0, (int)NULL},
{'j', "Increment the packet size every 'j' packets", 'i', NULL, 10, (int)NULL},
{'f', "Output file to log data in", 's', NULL, 0, "log.txt"},
{'a', "Audio file for SCO testing", 's', NULL, 0, "sound.raw"},
{'g', "Average similar sized results", 'b', NULL, FALSE, (int)NULL},
{'v', "Verbose", 'b', NULL, FALSE, (int)NULL}
};

unsigned char link_type;
bdaddr_t remote_bt_bdaddr;
int device_handle;
uint16_t connection_handle;
```

```
struct btan_log_data
{
        char link_type;
        char transmission_success;
        int packet_size;
        struct timeval echo_time;
        unsigned long bps;
        char link_quality;
        char rssi;
        char tx_pwr;
};

#define LOG_BUFFER_MAX_SIZE 500
struct btan_log_data log_buffer[LOG_BUFFER_MAX_SIZE];
static int log_buffer_index = 0;

/ ****************************************************************************************************
* Debug functions
****************************************************************************************************/
/ * Macros to help debugging */
#define verbose ((int)(COMMAND_LINE_OPTIONS[NR_OF_CMND_LINE_OPTIONS- 1].obtained_value_ptr))
#undef PDEBUG / * undef it, just in case */
#define PDEBUG(fmt, args...) if(verbose) printf("btan> " fmt, ## args)
static volatile int terminate = 0;

/ ***************************************************
* btan_sig_term()
*
* Functions for catching the termination signal for exiting gracefully
*
* int sig - the signal
*
* return - void
***************************************************/
static void btan_sig_term(int sig) {
        terminate = 1;
}

/ ***************************************************
* btan_print_buffer()
*
* Print the contents of a buffer / datapacket for debugging purpouses
*
* char* buffer - the data buffer
* int length - length of the buffer
*
* return - void
***************************************************/
void btan_print_buffer(char* buffer, int length)
{
        int i;
        if(verbose)
        {
                for(i=0;i<length;i++)
                {
                        printf("%02X ", (unsigned char)*(buffer+i));
                }
                printf("\n");
        }
return;
}

/ ****************************************************************************************************/

/ ***************************************************
* btan_delay()
*
* Delay functions to address timing issues
*
* unsigned short ticks - variable adjusting the amount of delay
*
* return - some number
***************************************************/
int btan_delay(unsigned short ticks)
{
        usleep(1000 * ticks);
        return 0;
}

/ ***************************************************
* btan_gen_buffer()
*
* Functions that generates random data to enter as payload into a data packet
*
* char* buffer - target buffer that should get the generated payload
* int size - size of the target payload
* int start - a seed that influences the generated contents
*
* return - number of bytes generated
***************************************************/
ssize_t btan_gen_buffer(char* buffer, int size, int start)
{
        int count;
        for(count=0; count<size; count++)
        {
                buffer[count] = (unsigned char)(((0xFA45 - (start & 0xFF)) + (count << 11)) * ((0x1234 + (start & 0xFF)) - (
                count << 6))) ^ (((start & 0xFF) + count) + (count << 2));
                if(start% 2)
                        buffer[count] = buffer[count] ^ (start & 0x55);
```

```
		else
			buffer[count] = buffer[count] ^ ((start >> 8) & 0xAA);
	}
	return count;
}

/ *************************************************
* btan_compare_buffers()
*
* Compare the contents of two buffers / data packets
*
* char* buf1 - the first buffer
* char* buf2 - the second buffer to compare with
* int length - the length on which comparison in
* the buffers should be done
*
* return - 0 on match, element number on mismatch
*************************************************/
int btan_compare_buffers(char* buf1, char* buf2, int length)
{
	int i;
	for(i=0;i<length;i++)
	{
		if(buf1[i] != buf2[i])
		{
			return i+1;
		}
	}
	return 0;
}

/ *************************************************
* btan_flush_log_buffer()
*
* Function to write logged data to file
*
* return - void
*************************************************/
void btan_flush_log_buffer(void)
{
	int i;
	char *filename;
	FILE *fd;

	/ * Open file */
	filename = (char*)(int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'f');
	PDEBUG("Opening file %s for writing\n", filename);
	fd = fopen(filename, "a");

	if(ftell(fd) == 0)
	{
		/ * This is a new file! */
		PDEBUG("Add column names to new file\n");
		fprintf(fd, "%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\t%s\n", "type", "success", "size", "bits", "sec", "usec", "bps",
		"rssi", "lq", "txpwr");
	}

	for(i=0;i<log_buffer_index;i++)
	{
		PDEBUG("Write element to file: %c\t%d\t%d\t%d\t%lu\t%ld\t%lu\t%d\t%d\t%d\n", log_buffer[i].link_type,
		log_buffer[i].transmission_success, log_buffer[i].packet_size, log_buffer[i].packet_size * 8, (unsigned long)log_buffer[i].
		echo_time.tv_sec, (unsigned long)log_buffer[i].echo_time.tv_usec, log_buffer[i].bps, (unsigned char)log_buffer[i].rssi, (
		fprintf(fd, "%d\t%d\t%d\t%d\t%ld\t%ld\t%lu\t%d\t%d\t%d\n", log_buffer[i].link_type, log_buffer[i].
		transmission_success, log_buffer[i].packet_size, log_buffer[i].packet_size * 8, (long)log_buffer[i].echo_time.tv_sec, (long)
		log_buffer[i].echo_time.tv_usec, log_buffer[i].bps, (unsigned char)log_buffer[i].rssi, (unsigned char)log_buffer[i].
	}

	/ * Close file*/
	fclose(fd);

	/ * Reset buffer */
	memset(log_buffer, 0, sizeof(struct btan_log_data) * LOG_BUFFER_MAX_SIZE);
	log_buffer_index = 0;
} « end btan_flush_log_buffer »

/ *************************************************
* btan_log_data()
*
* Function to handle logging of received data for link analysis
*
* char transmission_success - did the packet return intact?
* int pakcet_size - size of the packet sent
* struct timeval *echo_time - time it took for the packet to return
*
* return - void
*************************************************/
void btan_log_data(char transmission_success, int pakcet_size, struct timeval *echo_time)
{
	struct btan_log_data element;
	struct btan_log_data avg_val;
	static long avg_bps = 0, avg_success = 0, avg_sec = 0, avg_usec = 0, avg_lq = 0, avg_type = 0, avg_size = 0, avg_rssi = 0, avg_pwr = 0;
	static int count = 0;
	static int last_size = 0;
	static char average = 0;

	PDEBUG("Logging: %c\t%d\t%d\t%lu%lu\n", link_type, transmission_success, pakcet_size, (unsigned long)
	echo_time->tv_sec, (unsigned long)echo_time->tv_usec);
	element.link_type = link_type;
	element.transmission_success = transmission_success;
	element.packet_size = pakcet_size;
```

```
        element.echo_time.tv_sec = echo_time->tv_sec;
        element.echo_time.tv_usec = echo_time->tv_usec;
        element.bps = (unsigned long)((((double)pakcet_size * 8) * 1000000) / ((echo_time->tv_sec * 1000000) + echo_time->tv_usec));
        hci_read_link_quality(device_handle, connection_handle, &element.link_quality, 100);
        hci_read_rssi(device_handle, connection_handle, &element.rssi, 100);
        hci_read_transmit_power_level(device_handle, connection_handle, 0, &element.tx_pwr, 100);
        average = (char)(int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'g');
        if(average)
        {
                if(last_size == pakcet_size || last_size == 0)
                {
                        / * average this value */
                        avg_bps += element.bps;
                        avg_sec += element.echo_time.tv_sec;
                        avg_usec += element.echo_time.tv_usec;
                        avg_lq += element.link_quality;
                        avg_type += element.link_type;
                        avg_size += element.packet_size;
                        avg_rssi += element.rssi;
                        avg_pwr += element.tx_pwr;
                        avg_success += element.transmission_success;
                        count++;
                }
                else
                {
                        if(count > 0)
                        {
                                / * calculate average and log */
                                avg_val.bps = (unsigned long)(avg_bps / count);
                                avg_val.echo_time.tv_sec = (long)(avg_sec / count);
                                avg_val.echo_time.tv_usec = (long)(avg_usec / count);
                                avg_val.link_quality = (unsigned char)(avg_lq / count);
                                avg_val.link_type = (char)(avg_type / count);
                                avg_val.packet_size = (unsigned int)(avg_size / count);
                                avg_val.rssi = (unsigned char)(avg_rssi / count);
                                avg_val.tx_pwr = (unsigned char)(avg_pwr / count);
                                avg_val.transmission_success= (unsigned char)(avg_success / count);
                                count = 1;
                                avg_bps = element.bps;
                                avg_sec = element.echo_time.tv_sec;
                                avg_usec = element.echo_time.tv_usec;
                                avg_lq = element.link_quality;
                                avg_type = element.link_type;
                                avg_size = element.packet_size;
                                avg_rssi = element.rssi;
                                avg_pwr = element.tx_pwr;
                                avg_success = element.transmission_success;
                                log_buffer[log_buffer_index++] = avg_val;
                        } « end if count>0 »
                        else
                        {
                                log_buffer[log_buffer_index++] = element;
                        }
                } « end else »
                last_size = pakcet_size;
        } « end if average »
        else
        {
                log_buffer[log_buffer_index++] = element;
        }
        if(log_buffer_index >= LOG_BUFFER_MAX_SIZE)
        {
                btan_flush_log_buffer();
        }
        return;
} « end btan_log_data »

/ *************************************************
* btan_check_packet_sizes()
*
* Helper function to check if the requested packet length will fit in the
* Bluetooth packet type
*
* int length - the requested packet size
*
* return - 0 on success, negative on failure
*************************************************/
int btan_check_packet_sizes(int length)
{
        / * Check if we have space */
        if(length > MAXIMUM_BUFFER_SIZE)
        {
                printf("Error! Requested size %d is larger than buffer (%d)! \n", length, MAXIMUM_BUFFER_SIZE);
                return - 1;
        }
        switch(link_type)
        {
                case'l' :
                if(length > MAX_L2CAP_PACKET_SIZE)
                {
                        printf("Error! Requested size %d is larger than the maximum allowed (%d) for this type! \n", length,
                        MAX_L2CAP_PACKET_SIZE);
                        return - 1;
                }
                break;

                case'r' :
                if(length > MAX_RFCOMM_PACKET_SIZE)
                {
                        printf("Error! Requested size %d is larger than the maximum allowed (%d) for this type! \n", length,
                        MAX_RFCOMM_PACKET_SIZE);
                        return - 1;
```

```
            }
            break;
        }
        return 0;
} « end btan_check_packet_sizes »

/ ***************************************************
* btan_sco_exchange()
*
* Function to transfer SCO audio data over the SCO link
* for quality testing.
* Based on hstest.c by Marcel Holtmann
*
* int sco_socket - the sco socket
* int control_socket - the rfcomm socket for signalling control
* char mode - signify client or data connection to open
*
* return - 0 on success, negative on failure
***************************************************/
int btan_sco_exchange(int sco_socket, int control_socket, char mode)
{
        int sel;
        struct timeval timeout;
        fd_set rfds;
        unsigned char buffer[MAXIMUM_BUFFER_SIZE];
        int tx_length, rx_length = 0;
        bdaddr_t local;
        int local_device, maxfd;
        uint16_t vs;
        int fd;
        char *filename;
        mode_t filemode;
        int data_counter = 0;
        int error_count = 0;

        PDEBUG("Entering SCO Exchange in %c mode\n",mode);

        / * Check Audio settings */
        hci_devba(0, &local);
        local_device = hci_open_dev(0);
        hci_read_voice_setting(local_device, &vs, 1000);
        vs = htobs(vs);
        printf("Voice setting: 0x%04x\n", vs);
        close(local_device);
        if (vs ! = 0x0060)
        {
                printf("The voice setting must be 0x0060\n");
                return - 1;
        }
        switch (mode) {
                case 'c':
                filemode = O_RDONLY;
                break;
                case 's':
                filemode = O_WRONLY | O_CREAT | O_TRUNC;
                break;
                default:
                return - 1;
        }

        / * Open the input / output file */
        filename = (char*)(int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'a');
        PDEBUG("Opening file %s...\n", filename);
        if ((fd = open(filename, filemode)) < 0)
        {
                perror("Can't open input/ output file");
                return - 1;
        }
        if (mode == 's')
        write(control_socket, "RING\r\n", 6);
        maxfd = (control_socket > sco_socket) ? control_socket : sco_socket;
        PDEBUG("Transferring file...\n");
        printf("%08X", data_counter);fflush(stdout);
        while (! terminate)
        {
                FD_ZERO(&rfds);
                FD_SET(control_socket, &rfds);
                FD_SET(sco_socket, &rfds);
                timeout.tv_sec = 0;
                timeout.tv_usec = 10000;
                if ((sel = select(maxfd + 1, &rfds, NULL, NULL, &timeout)) > 0)
                {
                        if (FD_ISSET(sco_socket, &rfds))
                        {
                                printf("\b\b\b\b\b\b\b\b");
                                printf("%08X", data_counter);fflush(stdout);
                                memset(buffer, 0, sizeof(buffer));
                                rx_length = read(sco_socket, buffer, sizeof(buffer));
                                switch (mode)
                                {
                                        case 'c':
                                        rx_length = read(fd,buffer,rx_length);
                                        if(rx_length > 0)
                                        {
                                                tx_length = write(sco_socket, buffer ,rx_length);
                                                data_counter += tx_length;
                                        }
                                        else goto   exit;
                                        break;

                                        case 's':
```

```
                                if(rx_length <= 0)
                                {
                                        error_count++;
                                        if(error_count >= RX_ERROR_THRESHOLD)
                                        {
                                                terminate = 1;
                                        }
                                }
                                else
                                {
                                        data_counter += tx_length;
                                }
                                tx_length = write(fd, buffer,rx_length);
                                tx_length = write(sco_socket, buffer,rx_length);
                                break;
                        } « end switch mode »
                } « end if FD_ISSET(sco_socket,&... »
        } « end if (sel=select(maxfd+1,&... »
    } « end while ! terminate »
    exit:
    PDEBUG("\nExiting SCO exchange with data counter = %d and error_counter = %d\n", data_counter, error_count);
    close(fd);
} « end btan_sco_exchange »

/ **************************************************
* btan_client_data_pump()
*
* Function that generates and sends packets over the connection,
* then reads back the echoed packet and compares it with the
* original sent packet. The function then logs the roundtrip time
* and success rate of the echoed packet to be used in analysis.
*
* int socket - socket connection to use for data transmission
*
* return - 0 on success
**************************************************/
int btan_client_data_pump(int socket)
{
    static unsigned int packet_counter = 0, mismatch = 0;;
    int rx_length = 0, tx_length = 0;
    char read_buffer[MAXIMUM_BUFFER_SIZE];
    char write_buffer[MAXIMUM_BUFFER_SIZE];
    static unsigned int user_delay, user_counter, fake;
    struct timeval start_time, end_time, result_time;
    unsigned int user_packet_increment_counter, user_packet_increment_size;
    int read_size, current_l2cap_mtu;

    if(! verbose) {
        printf("%08X", packet_counter);fflush(stdout);
    }

    current_l2cap_mtu = btapi_l2cap_get_mtu(socket);
    user_delay = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'd');
    user_counter = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'n');
    tx_length = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'z');
    user_packet_increment_size = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'i');
    user_packet_increment_counter = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'j');

    / /while(READ_BUFFER[0] ! = MAGIC_CHARACTER)
    while((packet_counter < user_counter) && ! terminate)
    {
        if(packet_counter ! = 0)
        {
            if(! (packet_counter%user_packet_increment_counter))
            {
                / / PDEBUG("Increasing packet size from %d to %d\n", tx_length, tx_length + user_packet_increment_size);
                tx_length += user_packet_increment_size;
            }
        }
        packet_counter++;
        if(btan_check_packet_sizes(tx_length)) return - 1;

        / * Check MTU size */
        if((link_type == 'l') && (tx_length > current_l2cap_mtu))
        {
            PDEBUG("Increasing MTU size to %d\n", tx_length + 100);
            if(btapi_l2cap_set_mtu(socket, tx_length + 100)) return - 3;
            current_l2cap_mtu = tx_length + 100;
        }
        btan_gen_buffer(write_buffer, tx_length, packet_counter);
#ifdef PACKET_DEBUG
        PDEBUG("Sending message %d on socket %d:\n", packet_counter, socket);
        btan_print_buffer(write_buffer, tx_length);
#endif
        gettimeofday(&start_time, NULL);
        if(write(socket, write_buffer, tx_length) == - 1)
        {
            PDEBUG("Error writing message on socket %d! \n", socket);
            return - 1;
        }

        / * Read back the data */
        switch(link_type)
        {
            case 'l': read_size = MAX_L2CAP_PACKET_SIZE;break;
            case 'r': read_size = MAX_RFCOMM_PACKET_SIZE;break;
        }

        / * Check against buffer size */
        read_size = (read_size > MAXIMUM_BUFFER_SIZE) ? MAXIMUM_BUFFER_SIZE : read_size;
        PDEBUG("Reading data with buffer size set to %d\n", read_size);
```

```
                memset(read_buffer, 0, MAXIMUM_BUFFER_SIZE);
#ifdef BTAN_SERVER_ECHO
                rx_length = read(socket, read_buffer, MAXIMUM_BUFFER_SIZE);
#else
                memcpy(read_buffer, write_buffer, tx_length);
                rx_length = tx_length;
#endif
                gettimeofday(&end_time, NULL);
                result_time.tv_sec = end_time.tv_sec - start_time.tv_sec;
                result_time.tv_usec= end_time.tv_usec - start_time.tv_usec;
                if(result_time.tv_usec < 0)
                {
                        result_time.tv_sec- -;
                        result_time.tv_usec = 1000000  + result_time.tv_usec;
                }
                if(rx_length < 0)
                {
                        PDEBUG("Error reading message\n");
                        btan_log_data(0,tx_length, &result_time);
                }
                if(rx_length == 0)
                {
                        / / PDEBUG("Read back nothing! \n");
                        btan_delay(DELAY_MULTIPLIER);
                }
                if(rx_length > 0)
                {
                        if(rx_length ! = tx_length)
                        {
                                printf("\nMessage lengths do not match! \n");
#ifdef PACKET_DEBUG
                                printf("TX Buffer [%d]: ", tx_length);btan_print_buffer(write_buffer, tx_length);printf("\n");
                                printf("RX Buffer [%d]: ", rx_length);btan_print_buffer(read_buffer, rx_length);printf("\n");
#endif
                                printf("%08X", packet_counter);fflush(stdout);
                                btan_log_data(- 1,tx_length, &result_time);
                        }
                        else
                        {
                                if(mismatch = btan_compare_buffers(read_buffer, write_buffer, rx_length))
                                {
                                        printf("\nThe following buffer contents did not match on %d:\n", mismatch - 1);
#ifdef PACKET_DEBUG
                                        printf("TX Buffer [%d]: ", tx_length);btan_print_buffer(write_buffer, tx_length);
                                        printf("RX Buffer [%d]: ", rx_length);btan_print_buffer(read_buffer, rx_length);
#endif
                                        printf("%08X", packet_counter);fflush(stdout);
                                        btan_log_data(- 2,tx_length, &result_time);
                                }
                                else
                                {
                                        btan_log_data(1,tx_length, &result_time);
                                        if(verbose)
                                        {
                                                / / PDEBUG("Success with loopback time sec=%d and usec=%d\n", result_time.tv_sec, result_time.tv_usec);
                                        }
                                        else
                                        {
                                                printf("\b\b\b\b\b\b\b\b");
                                                printf("%08X", packet_counter);fflush(stdout);
                                        }
                                }
                        } « end else »
                } « end if rx_length>0 »
                fake = btan_delay(DELAY_MULTIPLIER * user_delay);
        } « end while (packet_counter<user_... »
        if(! verbose)
        {
                printf("\n");
        }

        / * wait a bit here to resolve some weird timing issue with large packets */
        sleep(1);

        / * Notify remote server of end */
        PDEBUG("Send quit sequence\n");
        write_buffer[0] = EXIT_SEQUENCE[0];
        write_buffer[1] = EXIT_SEQUENCE[1];
        write_buffer[2] = EXIT_SEQUENCE[2];
        write_buffer[3] = EXIT_SEQUENCE[3];
        write_buffer[4] = '\0';
        if(write(socket, write_buffer, 5) == - 1)
        {
                PDEBUG("Error writing message on socket %d! \n", socket);
                return - 1;
        }

        / * Flush the log buffer */
        btan_flush_log_buffer();
        / * long wait here to delay close for cases where UWB link is slower than BT */
        sleep(2);
        PDEBUG("Exiting data pump\n");
        return 0;
} « end btan_client_data_pump »

/ *************************************************
* btan_server_data_echo()
*
* Function that listens on a socket. Upon receival of a packet it
* is retransmitted back to the sender.
*
* int socket - socket connection to use for data transmission
```

```
*
* return - 0 on success
**************************************************/
int btan_server_data_echo(int socket)
{
        char read_buffer[MAXIMUM_BUFFER_SIZE];
        unsigned int packet_counter = 0;
        int num_bytes_read = 0;
        char quit = 0;
        int read_size;

        if(! verbose) {
                printf("%08X", packet_counter);fflush(stdout);
        }

        / * Read back the data */
        switch(link_type)
        {
                case 'l': read_size = MAX_L2CAP_PACKET_SIZE;break;
                case 'r': read_size = MAX_RFCOMM_PACKET_SIZE;break;
        }

        / * Check against buffer size */
        read_size = (read_size > MAXIMUM_BUFFER_SIZE) ? MAXIMUM_BUFFER_SIZE : read_size;
        PDEBUG("Starting echo loop with readsize %d...??\n", read_size);
        while(! quit && ! terminate)
        {
                packet_counter++;
                memset(read_buffer, 0, MAXIMUM_BUFFER_SIZE);
                num_bytes_read = read(socket, read_buffer, read_size);

                if(num_bytes_read < 0)
                {
                        PDEBUG("Error reading message on socket %d! \n", socket);
                }
                if(num_bytes_read == 0)
                {
                        btan_delay(DELAY_MULTIPLIER);
                        / / PDEBUG("Read back nothing! \n");
                }
                if(num_bytes_read > 0)
                {
                        if((read_buffer[0] == EXIT_SEQUENCE[0]) && (read_buffer[1] == EXIT_SEQUENCE[1]) && \
                        (read_buffer[2] == EXIT_SEQUENCE[2]) && (read_buffer[3] == EXIT_SEQUENCE[3]))
                        {
                                PDEBUG("Received quit! \n");
                                quit = 1;
                        }
                        else
                        {
#ifdef BTAN_SERVER_ECHO
                                if(write(socket, read_buffer, num_bytes_read) == - 1)
                                {
                                        PDEBUG("Error writing message on socket %d! \n", socket);
                                        return - 1;
                                }
#endif
                                if(verbose)
                                {
                                        PDEBUG("Echoing message %d on socket %d:\n", packet_counter, socket);
#ifdef PACKET_DEBUG
                                        btan_print_buffer(read_buffer, num_bytes_read);
#endif
                                }
                                else
                                {
                                        printf("\b\b\b\b\b\b\b\b");
                                        printf("%08X", packet_counter);fflush(stdout);
                                }
                        } « end else »
                } « end if num_bytes_read>0 »
        } « end while ! quit&&! terminate »
        PDEBUG("Exiting data echo\n");
        return 0;
} « end btan_server_data_echo »

/ *****************************************************************************************************/

/ **************************************************
* btan_run_server()
*
* Opens a server socket over the particular Bluetooth interface requested.
* Then branches to the echo function to relay received data back to the sender.
* char type - Signifies the requested Bluetooth interface (L2CAP / SCO / RFCOMM)
*
* return - 0 on success, negative on failure
**************************************************/
int btan_run_server(char type)
{
        int server, client;
        char info[BT_STR_ADDRESS_LENGTH] = { 0 };
        struct sockaddr_rc rem_rc_addr = { 0 };
        struct sockaddr_l2 rem_l2_addr = { 0 };
        struct sockaddr_sco rem_sco_addr = { 0 };
        socklen_t opt;
        struct sockaddr * rem_info;
        int channel = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'h');
        int fake;
        int sco_control, sco_control_server;

        / * We are in server mode*/
```

```
PDEBUG("Server mode\n");

/ * Continuous loop to accept incoming connections */
while(1)
{
        switch(type)
        {
                case'l' :
                        / * Open L2CAP Server Socket */
                        PDEBUG("Open L2CAP Server Socket\n");
                        if(channel == 0) channel = DEFAULT_L2CAP_PSM;
                        else channel += 2;
                        server = btapi_l2cap_open_server_socket(channel);
                        PDEBUG("L2CAP socket opened\n");
                        opt = sizeof(rem_l2_addr);
                        rem_info = (struct sockaddr *)&rem_l2_addr;
                break;

                case's' :
                        / * Open SCO Server Socket */
                        PDEBUG("Open SCO Server Socket\n");
                        server = btapi_sco_open_server_socket();
                        PDEBUG("SCO socket opened\n");
                        opt = sizeof(rem_sco_addr);
                        rem_info = (struct sockaddr *)&rem_sco_addr;
                break;

                case'r' :
                        / * Open RFCOMM Server Socket */
                        PDEBUG("Open RFCOMM Server Socket\n");
                        if(channel == 0) channel = DEFAULT_RFCOMM_CHANNEL;
                        else channel++;
                        server = btapi_rfcomm_open_server_socket(channel);
                        opt = sizeof(rem_rc_addr);
                        rem_info = (struct sockaddr *)&rem_rc_addr;
                break;

                default:
                        printf("Error! Invalid traffic type provided! \n");
                return - 2;
        } « end switch type »
        if(server <= 0)
        {
                printf("Error opening server socket! \n");
                return - 5;
        }

        / / printf("Waiting for connection on channel %d...\n", channel);
        client = accept(server, rem_info, &opt);
        if(client <= 0)
        {
                printf("Error accepting client on socket! \n");
                return - 2;
        }

        / * Convert socket info to show remote connection details */
        switch(type)
        {
                case'l' : ba2str( &rem_l2_addr.l2_bdaddr, info ); break;
                case's' : ba2str( &rem_sco_addr.sco_bdaddr, info ); break;
                case'r' : ba2str( &rem_rc_addr.rc_bdaddr, info); break;
        }

        fprintf(stderr, "Accepted connection from %s\n", info);
        if(type == 's')
        {
                sco_control_server = btapi_rfcomm_open_server_socket(DEFAULT_RFCOMM_CHANNEL);
                if(sco_control_server <= 0)
                {
                        printf("Error opening RFCOMM control server! \n");
                        return - 5;
                }
                opt = sizeof(rem_rc_addr);
                rem_info = (struct sockaddr *)&rem_rc_addr;
                sco_control = accept(sco_control_server, rem_info, &opt);
                btan_sco_exchange(client, sco_control, 's');

                close(sco_control);
                usleep(5);
                close(sco_control_server);
                usleep(5);
        }
        else btan_server_data_echo(client);

        / * shutdown client connection */
        / / PDEBUG("Shutdown client connection\n");
        / / shutdown(client, 0);
        sleep(1);

        / * close client connection */
        PDEBUG("Close client connection\n");
        close(client);
        if(type == 's')
        {
                / * SCO dont have recurring server connection */
                / * close server connection */
                PDEBUG("Close server connection\n");
                close(server);
                return 0;
        }
} « end while 1 »
```

```
	/ * close server connection */
	PDEBUG("Close server connection\n");
	close(server);
	return 0;
} « end btan_run_server »

/ ***************************************************
* btan_run_client()
*
* Opens a client socket over the particular Bluetooth interface requested
* to the provided Bluetooth remote address. Then branches to the datapump
* to exercise the link.
*
* char type - Signifies the requested Bluetooth interface (L2CAP / SCO / RFCOMM)
* char* remote_address - address of the remote Bluetooth device to connect to
*
* return - 0 on success, negative on failure
***************************************************/
int btan_run_client(char type, char* remote_address)
{
	bdaddr_t bdaddr;
	int client, sco_control;
	int channel = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'h');
	int dev_id, wait = 0;

	/ * We are in client mode */
	PDEBUG("Client mode\n");

	/ * Convert the provided address */
	if(btan_convert_bt_address(remote_address, &bdaddr))
	{
		printf("Error! Invalid address provided in 'client' mode! \n");
		return - 1;
	}
	dev_id = hci_get_route(NULL);
	device_handle = hci_open_dev(dev_id);
	connection_handle = find_conn(dev_id, &bdaddr);
	remote_bt_bdaddr = bdaddr;

	/ / wait = ((int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'z')) / 100;
	/ / if(wait <= 1) wait = 1;

	wait = 1;
	switch(type)
	{
		case'l' :
			/ * Open L2CAP Client Socket */
			PDEBUG("Open L2CAP Client Socket\n");
			if(channel == 0) channel = DEFAULT_L2CAP_PSM;
			client = btapi_l2cap_open_client_socket(bdaddr, channel);
		break;

		case's' :
			/ * Open SCO Client Socket */
			PDEBUG("Open SCO Client Socket\n");
			client = btapi_sco_open_client_socket(bdaddr);
		break;

		case'r' :
			/ * Open RFCOMM Client Socket */
			PDEBUG("Open RFCOMM Client Socket\n");
			if(channel == 0) channel = DEFAULT_RFCOMM_CHANNEL;
			client = btapi_rfcomm_open_client_socket(bdaddr, channel);
		break;

		default:
			printf("Error! Invalid traffic type provided! \n");
		return - 2;
	} « end switch type »

	if(client <= 0)
	{
		printf("Error opening client socket! \n");
		return - 5;
	}
	PDEBUG("Connected! \n\n");

	/ * Branch to data pump */
	if(type == 's')
	{
		sco_control = btapi_rfcomm_open_client_socket(bdaddr, DEFAULT_RFCOMM_CHANNEL);
		if(sco_control <= 0)
		{
			printf("Error opening RFCOMM control socket! \n");
			return - 5;
		}
		btan_sco_exchange(client, sco_control, 'c');
	}
	else
		{
		/ * Wait a little bit before starting */
		sleep(wait);
		btan_client_data_pump(client);
	}

	PDEBUG("Closing client socket\n");
	close(client);
	return 0;
} « end btan_run_client »

/ ***************************************************
* btan_convert_bt_address()
```

```
*
* Checks and converts a provided remote Bluetooth address to the correct format
*
* char* remote_address - the address in character array format
* bdaddr_t *bdaddr - the required binary format
*
* return - 0 on success
**************************************************/
int btan_convert_bt_address(char* remote_address, bdaddr_t *bdaddr)
{
        int count = 1;

        if(strlen(remote_address) ! = BT_STR_ADDRESS_LENGTH) return - 1;
        while(count < 6)
        {
                if(remote_address[(count*3) - 1] ! = ':') return - 2;
                count++;
        }
        return str2ba(remote_address, bdaddr);
}

/ **************************************************
* main()
*
* Main Entry point
* Parse the command line input to determine which
* Bluetooth analysis function to run, followed by the
* execution of this analysis.
*
* int argc - number of commandline argurments
* char* argv - the commandline argurments
*
* return - 0 on success
**************************************************/
int main(int argc, char* argv[])
{
        unsigned char* remote_address;
        int packet_length;
        struct sigaction sa;

        if(cmdline_parser_parse_commandline(argc, argv, COMMAND_LINE_OPTIONS))
        {
                cmdline_parser_print_usage(COMMAND_LINE_OPTIONS);
                return - 1;
        }
        if(verbose) cmdline_parser_print_options(COMMAND_LINE_OPTIONS);

        / * Check remote address */
        remote_address = (char*)(int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'c');
        link_type = (char)(int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 't');
        packet_length = (int)cmdline_parser_get_val(COMMAND_LINE_OPTIONS, 'z');
        if(btan_check_packet_sizes(packet_length)) return - 1;

        memset(&sa, 0, sizeof(sa));
        sa.sa_flags = SA_NOCLDSTOP;
        sa.sa_handler = btan_sig_term;
        sigaction(SIGTERM, &sa, NULL);
        sigaction(SIGINT, &sa, NULL);
        sa.sa_handler = SIG_IGN;
        sigaction(SIGCHLD, &sa, NULL);
        sigaction(SIGPIPE, &sa, NULL);

        if(remote_address[0] == (int)NULL)
        {
                return btan_run_server(link_type);
        }
        else
        {
                return btan_run_client(link_type, remote_address);
        }

        PDEBUG("Exiting application\n");
        return 0;
} « end main »
```

C UWB UTILITIES

This section gives a short description of the applied file-system based utilities provided with the "Linux UWB + Wireless USB + WiNET" driver stack needed to set up a UWB connection under Linux [23].

C.1 Connection Utilities

The `lsusb -v` command can be used for **probing** the connected UWB radio hardware together with `tail /var/log/messages` for inspecting the Linux UWB driver debug output.

For **scanning**, the driver offers the `scan` utility:

```
scan <CHANNEL-NUMBER> <TYPE>

Types:
   0 – scanning only
   1 – scanning outside the beacon period
   2 – scanning while inactive
   3 – scanning disabled
```

The `beacon` utility can be implemented to join a device to a **beacon group**:

```
beacon <CHANNEL-NUMBER> <BPST-OFFSET>
beacon <DEVADDR>        - beacon against that device's beacon group
                        - DEVADDR is a XX:YY 16 bit device address
```

After successfully loading UWB radio controllers on two different systems, the following two commands can be used to initiate beaconing (on channel 15 in this example) from the one device whilst scanning for beacons on the other:

```
PC_A$ echo 15 0 | cat > /sys/class/uwb_rc/uwb0/beacon
PC_B$ echo 15 0 | cat > /sys/class/uwb_rc/uwb0/scan
```

This should produce the following kernel debug messages on PC_B showing that a beacon was received from PC_A:

```
uwb device (mac 00:17:c4:0a:5c:56 dev a3:69) connected to usb 3-1:1.0
kernel: uwb-rc uwb0: wlp_uwb_notifs_cb: UWB device a3:69 is onair
```

The address of PC_A will also be added to the list of beaconing devices in the radio proximity in `/sys/bus/uwb/devices` , which will typically show the local and remote device as follows:

```
00:17:c4:0a:5c:56  ->  ../../../devices/pci0000:00/0000:00:1d.7/usb3/3-1/3-1:1.0/uwb0/00:17:c4:0a:5c:56
uwb0 -> ../../../devices/pci0000:00/0000:00:1d.7/usb3/3-1/3-1:1.0/uwb0
```

The current **DRP bandwidth availability** can be displayed by use of the `bw_avail` utility. Each of the 256 bits displayed in the bitmap value corresponds to an available MAS. The usage is as follows:

```
$ cat bw_avail
ffffffff,ffffffff,ffffffff,ffffffff,ffffffff,ffffffff,ffffffff,ffff0000
```

The **device** and **MAC addresses** of the UWB device can be set or queried by use of the `dev_address` and `mac_address` commands:

```
$ cat dev_address
5e:c0
$ cat mac_address
F9:80:f2:d1:5e:c0
```

Statistics about notifications received from the radio control device can be obtained by use of the `event_stats` utility.

For each connected device, say XX:YY, a sysfs filesystem is implemented to list device information:

`/sys/bus/uwb/devices/XX:YY/uwb:XX:YY/IEs`	**Beacon IEs** - shows the current Information Elements that the local or remote devices are transmitting in its beacon. The first byte is the IE ID, the next byte is the number of bytes in the IE, followed by that many bytes in the payload.
`/sys/bus/uwb/devices/XX:YY/uwb:XX:YY/LQE`	**Link Quality Estimate** – report the minimum, maximum and average link quality for the last 255 beacons received.
`/sys/bus/uwb/devices/XX:YY/uwb:XX:YY/RSSI`	**Radio Signal Strength Indication** – report the minimum, maximum and average RSSI for the last 255 beacons receive.

C.2 WLP Utilities

A **full discovery** of devices in the neighborhood is triggered by use of the `wlp_neighborhood` utility in `/sys/class/net/wlpX/`

The `wss_active` file can be used to

- Show all locally created **WSS** information
- Create a new **WSS**, i.e. `echo WSSID SECURE_STATUS ACCEPT_ENROLLMENT NAME | cat > /sys/class/net/wlpX/wss_activate`
- Enroll in and activate a **WSS** that has been learned about through discovery

For each **WSS** that has been activated, a new directory `/sys/class/net/wlpX/wss-<WSSID>` will be created, containing:

- A `properties` file listing the static properties of the WSS
- A `state` file listing the current state of the WSS
- A `members` file indicating the connected and unconnected neighbours

The **EDA cache** contains all neighbour connectivity information, and can be accessed by use of the `wlp_eda` command under `/sys/class/net/wlpX/`

The 16 byte **UUID** of the device can be viewed or set by the `wlp_uuid` utility, e.g.

```
$ echo 00 01 02 03 04 00 01 0 03 04 00 01 02 03 04 00 | cat >
/sys/class/ne/wlpX/wlp_uid
```

Under `/sys/class/net/wlpX/wlp_dev_XXXXXXX` the following utilities can be used to set **device information**:

Utility	Description	Size
`wlp_dev_name`	Contains the friendly name of the device	1-32 octets
`wlp_dev_manufacturer`	Name of the manufacturer	0-64 bytes
`wlp_dev_model_name`	Model name of the device	0-32 bytes
`wlp_dev_model_nr`	Model number of the device	0-32 bytes
`wlp_dev_serial`	Serial number of the device	0-32 bytes
`wlp_dev_prim_category`	Primary Device Type Category ID	
`wlp_dev_prim_OUI`	Primary Device Type OUI	
`wlp_dev_prim_OUI_sub`	Primary Device Type OUI Subdivision	
`wlp_dev_prim_subcat`	Primary Device Type Subcategory ID	

Table 26: WLP Device Information Utilities

The `uwb_phy_rate` utility can be used to set which **rate** the UWB PHY should use for transmitting data. The options are as follows:

Option	Rate
0	53.3 Mbps
1	80 Mbps

2	106 Mbps
3	160 Mbps
4	200 Mbps
5	320 Mbps
6	400 Mbps
7	480 Mbps

Table 27: UWB PHY Rate Options

The `uwb_ack_policy` command can be used to set or view to **ACK policy** for a WLP interface. The options are:

Option	**ACK Policy**
0	No ACK
1	Immediate ACK
2	B ACK
3	B REQ ACK

Table 28: UWB ACK Policy

RTS/CTS **flow control** in the UWB connection can be enabled or disabled by use of the `uwb_rts_cts` command.

The **PCA base priority** can be set on the fly by use of the `uwb_pca_base_priority` command using the following options:

Option	**Priority**	**Description**
0	AC_BE	Best Effort
1	AC_BK	Background
2	AC_BK	Background
3	AC_BE	Best Effort
4	AC_VI	Video
5	AC_VI	Video
6	AC_VO	Voice
7	AC_VO	Voice

Table 29: PCA Base Priority Options

The `wlp_lqe` command can be used to report the minimum, maximum and average **Line Quality Estimate** values in dB for the last 256 frames received.

The wlp_rssi command can be used to report the minimum, maximum and average **Received Signal Strength Indication** values in dBm for the last 256 frames received.

D BToUWB SYSTEM SOURCE FILES

D.1 UWB Router Kernel Module

This section lists the source code of the UWB Router module described in section 3.2.5.

D.1.1 UWB Router Kernel Module Header File

```
/****************************************************************
* UWB Router Control Kernel Module
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
****************************************************************
*
* This driver controls the data routing between upper layer Bluetooth
* modules and lower layer UWB device drivers.
*
* After setting up a WLP Ultra Wideband Link and loading the developed
* UWB WLP Convergence Layer Kernel Module, this module can be loaded
* to control data routing between upper layer Bluetooth modules and lower
* layer UWB driver and hardware. The uwb_router_downwards_routing()
* function performs the routing of data packets from higher layer like
* Bluetooth to the lower layer modules like UWB. Injecting data packets
* on the reverse path from UWB back into Bluetooth layers is performed by
* the uwb_router_upwards_routing() function.
*
****************************************************************/
#ifndef _UWB_ROUTER_H_
#define _UWB_ROUTER_H_

#define UWB_ROUTER_CONNECT_TO_BLUEZ 1
#define UWB_ROUTER_CONNECT_TO_UWB_WLP_CONV 1
#define GET_ROUTING_FROM_HEADER 1

/* Ioctl definitions */
#define UWB_ROUTER_IOC_MAGIC 'u'
#define UWB_LINK_CONTROL_MAGIC 'l'
#define UWB_ROUTER_GET_DRIVER_VERSION _IOR(UWB_ROUTER_IOC_MAGIC, 0, unsigned short)
#define UWB_ROUTER_GET_DEVICE_NAME _IOR(UWB_ROUTER_IOC_MAGIC, 1, char*)
#define UWB_ROUTER_GET_INS_MODE _IO(UWB_ROUTER_IOC_MAGIC, 2)
#define UWB_ROUTER_GET_ROUTING_MODE _IO(UWB_ROUTER_IOC_MAGIC, 3)
#define UWB_ROUTER_SET_INS_MODE _IOW(UWB_ROUTER_IOC_MAGIC, 4, unsigned char)
#define UWB_ROUTER_SET_ROUTING_MODE _IOW(UWB_ROUTER_IOC_MAGIC, 5, unsigned char)
#define UWB_ROUTER_REPORT_LINK_CONTROL_PRESENCE _IOW(UWB_ROUTER_IOC_MAGIC, 6, unsigned char)
#define UWB_ROUTER_IOC_MAXNR 7
#define UWB_ROUTING_MODE_DESCRIPTION_SIZE 50

struct uwb_routing_mode_descriptions{
        char identifier;
        char description[UWB_ROUTING_MODE_DESCRIPTION_SIZE];
};

enum UWB_ROUTING_MODE{
ROUTING_MODE_CLOSED = 0,
ROUTING_MODE_ETH_LINK,
ROUTING_MODE_UWB,
ROUTING_MODE_LOOPBACK,
ROUTING_MODE_END
} uwb_router_routing_mode;

struct uwb_routing_mode_descriptions uwb_router_routing_mode_description[ROUTING_MODE_END] = {
        {'n',"No Routing Mode\0"}
        ,{'p',"Straight Ethernet Passthrough Mode\0"}
        ,{'u',"UWB Hardware Mode\0"}
        ,{'l',"Local Loopback Mode\0"}
};

enum CHANNEL_INSERTION_MODE{
INS_MODE_TESTER = 0,
INS_MODE_L2CAP,
INS_MODE_SCO,
INS_MODE_END
} uwb_router_ins_mode;

struct uwb_routing_mode_descriptions uwb_router_ins_mode_description[INS_MODE_END] = {
        {'t',"UWB Controller Test Data\0"}
        ,{'c',"L2CAP Insertion Point\0"}
        ,{'s',"SCO Insertion Point\0"}
};

#if defined(__KERNEL__)
```

```
extern int uwb_router_downwards_routing(struct sk_buff *out_data, char ins_mode);
int uwb_router_upwards_routing(struct sk_buff *in_data);
#endif

#define UWB_ROUTER_HEADER_SIZE 4
#define UWB_ROUTER_ID 'T'

#endif / * _UWB_ROUTER_H_ */
```

D.1.2 UWB Router Kernel Module Source File

```
/ *************************************************************
* UWB Router Control Kernel Module
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/ or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
*************************************************************
*
* This driver controls the data routing between upper layer Bluetooth
* modules and lower layer UWB device drivers.
*
* After setting up a WLP Ultra Wideband Link and loading the developed
* UWB WLP Convergence Layer Kernel Module, this module can be loaded
* to control data routing between upper layer Bluetooth modules and lower
* layer UWB driver and hardware. The uwb_router_downwards_routing()
* function performs the routing of data packets from higher layer like
* Bluetooth to the lower layer modules like UWB. Injecting data packets
* on the reverse path from UWB back into Bluetooth layers is performed by
* the uwb_router_upwards_routing() function.
*
*************************************************************/
#include <linux/module.h>
#include <linux/ init.h>
#include <linux/ kernel.h> / * printk() */
#include <linux/ fs.h> / * everything... */
#include <linux/ cdev.h>
#include <linux/ poll.h>
#include <linux/ sched.h>
#include <linux/wait.h>
#include <asm/ uaccess.h> / * copy_*_user */
#include <asm/ delay.h>
#include <linux/ skbuff.h>
#include <linux/ net.h>
#include <net/ sock.h>
#include <linux/ tcp.h>
#include <linux/ in.h>
#include <asm/ uaccess.h>
#include <linux/ file.h>
#include <linux/ socket.h>
#include <linux/ smp_lock.h>
#include <linux/ slab.h>
#include "uwb_router_src.h"

/ ******************************************************************************************
* Module Red Tape
******************************************************************************************/
#ifndef UWB_ROUTER_MAJOR
#define UWB_ROUTER_MAJOR 0  / * dynamic major by default */
#endif
#ifndef UWB_ROUTER_NR_DEVS
#define UWB_ROUTER_NR_DEVS 1  / * uwb_router0 through uwb_router1 */
#endif

int uwb_router_major = UWB_ROUTER_MAJOR;
int uwb_router_minor = 0;
int uwb_router_nr_devs = UWB_ROUTER_NR_DEVS; / * number of bare scull devices */
static int uwb_router_use_count = 0;

module_param(uwb_router_major, int, S_IRUGO);
module_param(uwb_router_minor, int, S_IRUGO);
module_param(uwb_router_nr_devs, int, S_IRUGO);

MODULE_AUTHOR ("Etienne van der Linde");
MODULE_DESCRIPTION("UWB Router");
MODULE_LICENSE("GPL");

/ * Split minors in two parts */
#define TYPE(minor) (((minor) >> 4) & 0xf) / * high nibble */
#define NUM(minor) ((minor) & 0xf) / * low nibble */

/ * The different configurable parameters */
extern int uwb_router_major; / * main.c */
extern int uwb_router_nr_devs;

#define NAME_SIZE 30
#define UWB_ROUTER_DRIVER_VERSION 5

struct uwb_router_defaults{
        char device_name[NAME_SIZE];
};

struct uwb_router_defaults DEFAULTS_ARRAY[UWB_ROUTER_NR_DEVS] = {
        {"UWB ROUTER\0"}
};

/ * Common device private data */
struct uwb_router_dev {
        uint8_t local_id; / * local identifier */
        char* device_name; / * local identifier */
        struct semaphore sem; / * mutual exclusion semaphore */
        struct cdev cdev; / * Char device structure */
};
#define setup_uwb_router_dev(dev,index) \
dev->local_id = (uint8_t)index; \
dev->device_name = DEFAULTS_ARRAY[index].device_name;

struct uwb_router_dev *uwb_router_devices; / * allocated in uwb_router_init_module */

/ * Prototypes for shared functions */
ssize_t uwb_router_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos);
ssize_t uwb_router_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);
int uwb_router_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);

/ ***********************************************************************************
* Debug Printing
***********************************************************************************/
#define UWB_ROUTER_MODULE_DEBUG 1
/ /#define UWB_ROUTER_PACKET_DEBUG 1

/ * Macros to help debugging */

/ * Module specific debug printing */
#undef MDEBUG / * undef it, just in case */
#ifdef UWB_ROUTER_MODULE_DEBUG
# define MDEBUG(fmt, args...) printk(">>uwb_wlp_conv: " fmt, ## args)
#else
# define MDEBUG(fmt, args...) / * not debugging: nothing */
#endif

/ * Packet specific debug printing */
#undef PDEBUG / * undef it, just in case */
#ifdef UWB_ROUTER_PACKET_DEBUG
# define PDEBUG(fmt, args...) printk(">>uwb_router: " fmt, ## args)
#else
# define PDEBUG(fmt, args...) / * not debugging: nothing */
#endif

/ ***********************************************************************************
* External module connections
***********************************************************************************/

static char uwb_link_control_present = 0;
struct fasync_struct *uwb_router_async_queue; / * asynchronous readers */

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
extern void l2cap_over_uwb_set_routing_mode(char mode);
extern int l2cap_over_uwb_register_transmitter_function(void* transmitter_function);
extern int l2cap_over_uwb_bottom_entry(struct sk_buff *RxSkb);
extern void sco_over_uwb_set_routing_mode(char mode);
extern int sco_over_uwb_register_transmitter_function(void* transmitter_function);
extern int sco_over_uwb_bottom_entry(struct sk_buff *RxSkb);
#endif

#ifdef UWB_ROUTER_CONNECT_TO_UWB_WLP_CONV
#include "../ uwb_wlp_conv/ uwb_wlp_conv_src.h"
#endif
/ ***********************************************************************************
* Data packet control and routing
***********************************************************************************/
struct uwb_router_header
{
        __u8 id;
        __u16 len;
        __u8 mode;
}__attribute__ ((packed));

static struct sk_buff *rx_head, *rx_tail;

/ ***********************************************************************************
* Debug Functions
***********************************************************************************/

/ **************************************************
* uwb_router_print_skb()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct sk_buff *skb_ptr - structure containing the data
*
* return - number of bytes printed
```

```
**************************************************/
ssize_t uwb_router_print_skb(struct sk_buff *skb_ptr)
{
	int i = 0;

#ifdef UWB_ROUTER_PACKET_DEBUG
	for(i=0; i<skb_ptr->len; i++)
	{
		printk("%02X ", (unsigned char)(skb_ptr->data[i]));
	}
	printk("\n");
#endif

	return i;
}

/ ***************************************************************************************
* Buffer Routing Functions
***************************************************************************************/

/ **************************************************
* uwb_router_put_header()
*
* Adds a header to the data packet to transport routing information
* to the receiver.
*
* struct sk_buff *buff_data - structure containing the data buffer
* char mode - buffer containing the data
*
* return - always 0 for now
**************************************************/
int uwb_router_put_header(struct sk_buff *buff_data, char mode)
{
	struct uwb_router_header *hdr;

	hdr = (struct uwb_router_header *) skb_push(buff_data, UWB_ROUTER_HEADER_SIZE);
	hdr->id = UWB_ROUTER_ID;
	hdr->len = __cpu_to_le16((unsigned short) (buff_data->len - UWB_ROUTER_HEADER_SIZE));
	hdr->mode = mode;

	return 0;
}

/ **************************************************
* uwb_router_detach_packet()
*
* Detaches trailing packets from a multi- packet buffer
*
* struct sk_buff *buff_data - structure containing the multiple packet buffer
* int index - index in the buffer where the next packet resides
*
* return - 0 on success
**************************************************/
void uwb_router_detach_packet(struct sk_buff *buff_data, int index)
{
	struct sk_buff *extra_buffer = 0;

	PDEBUG("Detaching packet at index %d\n", index);
	extra_buffer = alloc_skb(buff_data->len - index, GFP_ATOMIC);
	skb_put(extra_buffer, buff_data->len - index);
	memcpy(extra_buffer->data, &(buff_data->data[index]), buff_data->len - index);
	extra_buffer->sk = buff_data->sk;
	buff_data->len = index;
	buff_data->next = extra_buffer;
}

/ **************************************************
* uwb_router_pull_header()
*
* Removes the header from data packet
*
* struct sk_buff *buff_data - structure containing the data buffer
*
* return - routing information embedded in header
**************************************************/
char uwb_router_pull_header(struct sk_buff *buff_data)
{
	struct uwb_router_header *hdr = (struct uwb_router_header *) buff_data->data;
	char mode = hdr->mode;
	int len = hdr->len;

	if(hdr->id != UWB_ROUTER_ID)
	return - 1;

	skb_pull(buff_data, UWB_ROUTER_HEADER_SIZE);
	PDEBUG("header_len = %d vs buf_len = %d\n", len, buff_data->len);
	if(len ! = buff_data->len)
	{
		if(buff_data->data[len] == UWB_ROUTER_ID)
		{
			/ * There might be another packet attached! */
			uwb_router_detach_packet(buff_data, len);
		}
		else
		{
			return - 2;
		}
	}
	return mode;
} « end uwb_router_pull_header »
```

```
/ **************************************************
* uwb_router_upwards_routing()
*
* Extracts the header from the data packet and then passes it on to
* the interface defined in this header
*
* struct sk_buff *in_data - structure containing the data buffer
*
* return - 0 on success
**************************************************/
int uwb_router_upwards_routing(struct sk_buff *in_data)
{
        char ins_mode;
        struct sk_buff * rx_new;

        ins_mode = uwb_router_pull_header(in_data);
        PDEBUG("Routing packet of size %d upwards to %d:\n", in_data->len, ins_mode);
        uwb_router_print_skb(in_data);
        switch(ins_mode)
        {
                case INS_MODE_L2CAP: / * send packet to L2CAP */
#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
                    PDEBUG("Routing packet of size %d to INS_MODE_L2CAP\n", in_data->len);
                    l2cap_over_uwb_bottom_entry(in_data);
#else
                    PDEBUG("Routing packet to INS_MODE_L2CAP - not! \n");
#endif
                break;

                case INS_MODE_SCO: / * send packet to SCO */
#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
                        PDEBUG("Routing packet to INS_MODE_SCO\n");

                        / * Perhaps add synchronous retiming here??*/
                        / /mdelay(500);
                        sco_over_uwb_bottom_entry(in_data);
#else
                        PDEBUG("Routing packet to INS_MODE_SCO - not! \n");
#endif
                break;

                case INS_MODE_TESTER: / * send packet to Tester */

                        if(uwb_link_control_present == UWB_LINK_CONTROL_MAGIC)
                        {
                                PDEBUG("Routing packet to INS_MODE_TESTER\n");
                                rx_new = skb_clone(in_data, GFP_ATOMIC);

                                if(rx_tail) / / the list is not empty
                                {
                                        rx_tail->next = rx_new; / / link it in
                                }
                                else / / the list is empty
                                {
                                        rx_head = rx_new;
                                }
                                rx_tail = rx_new; / / advance the list
                                kill_fasync(&uwb_router_async_queue, SIGIO, POLL_IN);
                        }
                        else
                        {
                                PDEBUG("Packet received but no user program registered to send it to! \n");
                        }
                break;

                default:
                break;
        } « end switch ins_mode »

        / * Process attached packet */
        if(in_data->next)
        {
                uwb_router_upwards_routing(in_data->next);
                kfree_skb(in_data->next);
        }
        return 0;
} « end uwb_router_upwards_routing »

/ **************************************************
* uwb_router_downwards_routing()
*
* Adds a header to the data packet and then passes it on to
* a user defined interface.
*
* struct sk_buff *out_data - structure containing the data buffer
* char ins_mode - signifies where data originates from,
* to be used in upwards routing on return path
*
* return - return value of handling function, else negative
**************************************************/
int uwb_router_downwards_routing(struct sk_buff *out_data, char ins_mode)
{
        PDEBUG("Routing data downwards from %d to %d\n", ins_mode, uwb_router_routing_mode);
        uwb_router_print_skb(out_data);

        if(uwb_router_put_header(out_data, ins_mode))
        {
                PDEBUG("Error adding header! \n");
                return - 2;
        }

        PDEBUG("After adding header:\n");
```

```
	uwb_router_print_skb(out_data);

	switch(uwb_router_routing_mode)
	{
		case ROUTING_MODE_UWB: / * invoke wlp_link driver */
			PDEBUG("Routing packet of size %d to ROUTING_MODE_UWB! \n", out_data->len);
#ifdef UWB_ROUTER_CONNECT_TO_UWB_WLP_CONV
			return uwb_wlp_conv_tx_packet_top_entry(out_data) - UWB_ROUTER_HEADER_SIZE;
#endif
		break;

		case ROUTING_MODE_LOOPBACK: / * Route packet back upwards */
			PDEBUG("Routing packet to ROUTING_MODE_LOOPBACK...\n");
			return uwb_router_upwards_routing(out_data);
		break;

		default:
		break;
	}
	return - 1;
} « end uwb_router_downwards_routing »

/ ****************************************************************************************************
* Device IO Functions
****************************************************************************************************/

/ ***************************************************
* uwb_router_read()
*
* Interface for passing a buffer of data up to the
* user space program for reading.
* Wakeup the asyncronous wait queue to indicated
* that data has been serviced
*
* struct file *filp - linked list device container
* const char __user *buf - buffer containing the data
* size_t count - number of bytes in buffer
* loff_t *f_pos - position pointer of current data
*
* return - number of bytes read from device
***************************************************/
ssize_t uwb_router_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
	struct uwb_router_dev *dev = filp->private_data;
	ssize_t retval = 0;
	int current_size = 0;
	struct sk_buff *holder;

	if (down_interruptible(&dev->sem))
		return - ERESTARTSYS;

	if(rx_head) / / there is data in the list
	{
		current_size = rx_head->len;
		PDEBUG("Entering uwb_router_read with count %d and current size %d\n", count, current_size);
		if(count < current_size)
			current_size = count;
		PDEBUG("uwb_router_read: Sending buffer upwards [%d]:\n", current_size);
		uwb_router_print_skb(rx_head);
		memcpy(buf, rx_head->data, current_size);
		retval = current_size;
		skb_pull(rx_head, current_size);
		if(rx_head->len <= 0) / / done with this buffer
		{
			/ *advance the list */
			holder = rx_head;
			if(rx_head->next) / / there is something in the list
			{
				rx_head = rx_head->next;
			}
			else / / the list is empty again - clean up!
			{
				if(rx_head == rx_tail)
				{
					rx_tail = NULL;
				}
				rx_head = NULL;
			}

			/ * free the current buffer */
			kfree_skb(holder);
		}
	} « end if rx_head »
	up(&dev->sem);
	return retval;
} « end uwb_router_read »

/ ***************************************************
* uwb_router_write()
*
* Interface for passing a buffer of data down to the
* driver for writing.
*
* struct file *filp - linked list device container
* const char __user *buf - buffer containing the data
* size_t count - number of bytes in buffer
* loff_t *f_pos - position pointer of current data
*
* return - number of bytes written
***************************************************/
ssize_t uwb_router_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos)
{
        struct uwb_router_dev *dev = filp->private_data;
        ssize_t retval = -ENOMEM; / * value used in "goto out" statements */
        struct sk_buff *skb_out;

        PDEBUG("Entering uwb_router_write\n");
        if (down_interruptible(&dev->sem))
                return - ERESTARTSYS;

        / * Insert data into skb structure */
        skb_out = alloc_skb(count + UWB_ROUTER_HEADER_SIZE, GFP_ATOMIC);
        skb_reserve(skb_out, UWB_ROUTER_HEADER_SIZE); / * Reserve space for header */
        memcpy(skb_put(skb_out, count), buf, count);

        PDEBUG("Driver received buffer data: ");
        uwb_router_print_skb(skb_out);

        / * use global routing variables */
        retval = uwb_router_downwards_routing(skb_out, INS_MODE_TESTER);

        PDEBUG("Done! Freeing buffer...\n");
        / * free the allocated buffer */
        kfree_skb(skb_out);
        up(&dev->sem);

        return retval;
} « end uwb_router_write »

/ *************************************************
* uwb_router_ioctl()
*
* IO Control function to access variables from user space.
*
* struct inode *inode -
* struct file *filp - linked list device container
* unsigned int cmd - command index
* unsigned long arg - passing data
*
* return - 0 for success,
**************************************************/
int uwb_router_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
{
        struct uwb_router_dev *dev = filp->private_data;
        int err = 0;
        int retval = 0;
        char* dev_name= 0;

        MDEBUG("Entering uwb_router_ioctl\n");
        /*
        * extract the type and number bitfields, and don't decode
        * wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
        */
        if (_IOC_TYPE(cmd) ! = UWB_ROUTER_IOC_MAGIC) return - ENOTTY;
        if (_IOC_NR(cmd) > UWB_ROUTER_IOC_MAXNR) return - ENOTTY;
        /*
        * the direction is a bitmask, and VERIFY_WRITE catches R/W
        * transfers. `Type' is user- oriented, while
        * access_ok is kernel- oriented, so the concept of "read" and
        * "write" is reversed
        */
        if (_IOC_DIR(cmd) & _IOC_READ)
                err = ! access_ok(VERIFY_WRITE, (void __user *)arg, _IOC_SIZE(cmd));
        else if (_IOC_DIR(cmd) & _IOC_WRITE)
                err = ! access_ok(VERIFY_READ, (void __user *)arg, _IOC_SIZE(cmd));
        if (err) return - EFAULT;

        switch(cmd) {
                case UWB_ROUTER_GET_DRIVER_VERSION:
                        retval = __put_user (UWB_ROUTER_DRIVER_VERSION, (unsigned short __user *) arg);
                break;

                case UWB_ROUTER_GET_DEVICE_NAME:
                        MDEBUG("Retrieving device name %s\n",dev->device_name);
                        dev_name = (char*) arg;
                        memcpy(dev_name, dev->device_name, strlen(dev->device_name)+1);
                        retval = strlen(dev->device_name);
                break;

                case UWB_ROUTER_GET_INS_MODE:
                        retval = uwb_router_ins_mode;
                break;

                case UWB_ROUTER_GET_ROUTING_MODE:
                        retval = uwb_router_routing_mode;
                break;

                case UWB_ROUTER_SET_INS_MODE:
                        retval = __get_user (uwb_router_ins_mode, (char __user *) arg);
                        MDEBUG("Set INS Mode to %s\n", uwb_router_ins_mode_description[uwb_router_ins_mode].description);
#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
                        if(uwb_router_ins_mode == INS_MODE_L2CAP)
                        {
                                l2cap_over_uwb_set_routing_mode(1);
                        }
                        else
                        {
                                l2cap_over_uwb_set_routing_mode(0);
                        }
                        if(uwb_router_ins_mode == INS_MODE_SCO)
                        {
                                sco_over_uwb_set_routing_mode(1);
                        }
                        else
                        {
                                sco_over_uwb_set_routing_mode(0);
                        }
#endif
                break;

                case UWB_ROUTER_SET_ROUTING_MODE:
                        retval = __get_user (uwb_router_routing_mode, (char __user *) arg);
                        MDEBUG("Set Routing Mode to %s\n", uwb_router_routing_mode_description[uwb_router_routing_mode].
                        description);
                break;

                case UWB_ROUTER_REPORT_LINK_CONTROL_PRESENCE:
                        retval = __get_user (uwb_link_control_present, (char __user *) arg);
                        MDEBUG("Set uwb_link_control_present to %02X\n", uwb_link_control_present);
                break;

                default: / * redundant, as cmd was checked against MAXNR */
                return - ENOTTY;
        } « end switch cmd »

        return retval;
} « end uwb_router_ioctl »

/ **************************************************
* uwb_router_fasync()
*
* Asyncronous notifier to userspace program.
*
* int fd - file descriptor
* struct file *filp - linked list device container
* int mode -
*
* return - positive for success,
* negative for failure
**************************************************/
static int uwb_router_fasync(int fd, struct file *filp, int mode)
{
        return fasync_helper(fd, filp, mode, &uwb_router_async_queue);
}

/ ************************************************************************************************
* Standard device operations
************************************************************************************************/

/ **************************************************
* uwb_router_open()
*
* Operations to perform when the device is opened.
*
* struct inode *inode -
* struct file *filp - linked list device container
*
* return - always 0
**************************************************/
int uwb_router_open(struct inode *inode, struct file *filp)
{
        struct uwb_router_dev *dev; / * device information */

        MDEBUG("Entering uwb_router_open\n");
        dev = container_of(inode->i_cdev, struct uwb_router_dev, cdev);
        filp->private_data = dev; / * for other methods */
        uwb_router_use_count++;
        return 0; / * success */
}

/ **************************************************
* uwb_router_release()
*
* Operations to perform when the device is closed.
*
* struct inode *inode -
* struct file *filp - linked list device container
*
* return - always 0
**************************************************/
int uwb_router_release(struct inode *inode, struct file *filp)
{
        MDEBUG("Entering uwb_router_release\n");
        uwb_router_fasync(- 1, filp, 0);
        uwb_router_use_count- -;
        return 0;
}

struct file_operations uwb_router_fops = {
        .owner = THIS_MODULE,
        .read = uwb_router_read,
        .write = uwb_router_write,
        .ioctl = uwb_router_ioctl,
        .open = uwb_router_open,
        .release = uwb_router_release,
        .fasync = uwb_router_fasync,
};

/ **************************************************
* uc_print_values()
*
* Print private structure data values for debugging.
*
* struct uwb_router_dev *dev - the device private data
*
```

```
* return - void
**************************************************/
static void uc_print_values(struct uwb_router_dev *dev)
{
	MDEBUG("UWB Controller %d values:\n", dev->local_id);
	MDEBUG("Driver Version\t= %04X\n", UWB_ROUTER_DRIVER_VERSION);
	MDEBUG("Device Name\t= %s\n", dev->device_name);
}

/ **************************************************
* uwb_router_setup_cdev()
*
* Set up the char_dev structure for this device.
*
* struct uwb_router_dev *dev - the device private data
* int index - the device indexing number
*
* return - integer value from cdev_add
**************************************************/
static int uwb_router_setup_cdev(struct uwb_router_dev *dev, int index)
{
	int err = 0;
	dev_t devno = MKDEV(uwb_router_major,uwb_router_minor + index);

	MDEBUG("Entering uwb_router_setup_cdev for device %d\n", index);
	MDEBUG("Setting up default struct values...\n");
	setup_uwb_router_dev(dev, index);
	MDEBUG("Registering node with minor number %d\n", index);
	cdev_init(&dev->cdev, &uwb_router_fops);
	dev->cdev.owner = THIS_MODULE;
	kobject_set_name(&(dev->cdev.kobj), "uwb_router");
	dev->cdev.ops = &uwb_router_fops;
	MDEBUG("Performing the dangerous add...\n");

	/ * if this fails - there's something wrong with make! */
	err = cdev_add (&dev->cdev, devno, 1);

	/ * Fail gracefully if need be */
	if (err)
	{
		printk(KERN_NOTICE "Error %d adding uwb_router%d", err, index);
	}
	MDEBUG("All done with uwb_router_setup_cdev...\n");
	return err;
} « end uwb_router_setup_cdev »

/ **************************************************
* uwb_router_exit()
*
* Standard operations to remove device driver from system.
*
* void
*
* return - void
**************************************************/
static void __exit uwb_router_exit(void)
{
	int i;
	dev_t devno = MKDEV(uwb_router_major, uwb_router_minor);

	MDEBUG("Entering uwb_router_exit\n");
	printk(KERN_INFO "Exiting UWB Controller Module\n");

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
	/ * Remove receiving functions from Bluetooth stack */
	l2cap_over_uwb_register_transmitter_function(NULL);
	sco_over_uwb_register_transmitter_function(NULL);
#endif

#ifdef UWB_ROUTER_CONNECT_TO_UWB_WLP_CONV
	/ * Remove receiving function from UWB WLP Convergence layer */
	uwb_wlp_conv_register_receiver_function(NULL);
#endif

	/ * Get rid of our char dev entries */
	if (uwb_router_devices) {
		for (i = 0; i < uwb_router_nr_devs; i++) {
			cdev_del(&uwb_router_devices[i].cdev);
		}
		kfree(uwb_router_devices);
	}
	/ * cleanup_module is never called if registering failed */
	unregister_chrdev_region(devno, uwb_router_nr_devs);
	MDEBUG("Leaving.\n");
} « end uwb_router_exit »

/ **************************************************
* uwb_router_init()
*
* Standard operations to setup and add device driver to system.
*
* void
*
* return - 0 if success
**************************************************/
static int __init uwb_router_init(void)
{
	int result, i;
	dev_t dev = 0; / /MKDEV(UWB_ROUTER_MAJOR, 0);

	MDEBUG("Entering uwb_router_init\n");
```

```
        MDEBUG("Initializing UWB Controller Module...\n");

        if (uwb_router_major) {
                dev = MKDEV(uwb_router_major, uwb_router_minor);
                result = register_chrdev_region(dev, uwb_router_nr_devs, "uwb_router");
        } else {
                result = alloc_chrdev_region(&dev, uwb_router_minor, uwb_router_nr_devs, "uwb_router");
                uwb_router_major = MAJOR(dev);
        }
        if (result < 0) {
                printk(KERN_WARNING "uwb_router: can't get major %d\n", uwb_router_major);
                return result;
        }
        MDEBUG("Module registered with major device number %d\n", uwb_router_major);
        uwb_router_devices = kmalloc(uwb_router_nr_devs * sizeof(struct uwb_router_dev), GFP_KERNEL);
        if (! uwb_router_devices) {
                result = -ENOMEM;
                goto fail; / * Make this more graceful */
        }
        memset(uwb_router_devices, 0, uwb_router_nr_devs * sizeof(struct uwb_router_dev));

        / * Initialize each device. */
        for (i = 0; i < uwb_router_nr_devs; i++) {
                init_MUTEX(&uwb_router_devices[i].sem);
                uwb_router_devices[i].cdev.dev = dev;
                if(uwb_router_setup_cdev(&uwb_router_devices[i], i) ! = 0)
                goto fail;
                uc_print_values(&uwb_router_devices[i]);
        }

        / * Initialize the asynchronous queue for passing received data to user space */
        uwb_router_async_queue = kmalloc(sizeof(struct fasync_struct), GFP_KERNEL);
        memset(uwb_router_async_queue, 0, sizeof(struct fasync_struct));
        uwb_router_async_queue = NULL;

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
        / * Register receiving functions with Bluetooth stack */
        l2cap_over_uwb_register_transmitter_function((void*)uwb_router_downwards_routing);
        sco_over_uwb_register_transmitter_function((void*)uwb_router_downwards_routing);
#endif

#ifdef UWB_ROUTER_CONNECT_TO_UWB_WLP_CONV
        / * Register receiving function with UWB WLP Convergence layer */
        uwb_wlp_conv_register_receiver_function((void*)uwb_router_upwards_routing);
#endif

        return 0;
        fail:
        uwb_router_exit();
        return result;
} « end uwb_router_init »

module_init(uwb_router_init);
module_exit(uwb_router_exit);
```

D.2 UWB WLP Convergence Module

This section lists the source code of the UWB WLP Convergence module described in section 3.2.6.

D.2.1 UWB WLP Convergence Module Header File

```
/ ***************************************************************
* UWB WLP Convergence Layer Kernel Module
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/ or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
***************************************************************
*
* This driver acts as interface between upper layer router / Bluetooth
* modules and lower layer UWB device drivers.
*
* After setting up a WLP Ultra Wideband Link, the module can be loaded
* with the target device WLP IP address as module parameter.
* The driver will:
*
* a) Create a thread to attempt opening a socket over WLP to this target
* UWB device to be used for transmission.
* b) Create a thread for a server which will listen for incoming
* connections from remote UWB hardware to be used for receiving
* data from the remote UWB based device.
*
* After establishing a connection to the remote WLP UWB based device,
* the uwb_wlp_conv_tx_packet_top_entry() function may be used to
* transmit packets over this link, and the
* uwb_wlp_conv_register_receiver_function() function may be used to
* regtister a calling function to handle packets received over this link.
*
***************************************************************/
#ifndef _UWB_WLP_CONV_H_
#define _UWB_WLP_CONV_H_

#define UWB_WLP_CONV_DRIVER_VERSION "1.7"

/ * Ioctl definitions */
#define UWB_WLP_CONV_IOC_MAGIC 'u'
#define UWB_WLP_CONV_GET_DRIVER_VERSION _IOR(UWB_WLP_CONV_IOC_MAGIC, 0, unsigned short)
#define UWB_WLP_CONV_GET_STATUS _IOR(UWB_WLP_CONV_IOC_MAGIC, 1, unsigned short)
#define UWB_WLP_CONV_GET_DEVICE_NAME _IOR(UWB_WLP_CONV_IOC_MAGIC, 2, char*)
#define UWB_WLP_CONV_IOC_MAXNR 3

extern int uwb_wlp_conv_tx_packet_top_entry(struct sk_buff *rx_buff);
extern int uwb_wlp_conv_register_receiver_function(void* receiver_function);
int uwb_wlp_conv_rx_packet_top_exit(struct sk_buff *rx_buff);

#endif / * _UWB_WLP_CONV_H_ */
```

D.2.2 UWB WLP Convergence Module Source File

```
/ ***************************************************************
* UWB WLP Convergence Layer Kernel Module
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/ or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
***************************************************************
*
* This driver acts as interface between upper layer router / Bluetooth
* modules and lower layer UWB device drivers.
*
* After setting up a WLP Ultra Wideband Link, the module can be loaded
* with the target device WLP IP address as module parameter.
* The driver will:
*
* a) Create a thread to attempt opening a socket over WLP to this target
* UWB device to be used for transmission.
* b) Create a thread for a server which will listen for incoming
* connections from remote UWB hardware to be used for receiving
* data from the remote UWB based device.
*
* After establishing a connection to the remote WLP UWB based device,
* the uwb_wlp_conv_tx_packet_top_entry() function may be used to
* transmit packets over this link, and the
* uwb_wlp_conv_register_receiver_function() function may be used to
* regtister a calling function to handle packets received over this link.
*
***************************************************************/

#include <linux/ net.h>
#include <net/ sock.h>
#include <linux/ tcp.h>
#include <linux/ in.h>
#include <asm/ uaccess.h>
#include <linux/ file.h>
#include <linux/ socket.h>
#include <linux/ smp_lock.h>
#include <linux/ slab.h>
#include <linux/ delay.h>
#include <linux/ byteorder/ generic.h>
#include <linux/module.h>
#include <linux/ init.h>
#include <linux/ kernel.h> / * printk() */
#include <linux/ fs.h> / * everything... */
#include <linux/ cdev.h>
#include <asm/ uaccess.h> / * copy_*_user */
#include <asm/ delay.h>
#include "uwb_wlp_conv_src.h"
/ /#include "../ uwb_router/ uwb_router_src.h"

/ ***************************************************************************************************
* Module Red Tape
***************************************************************************************************/

#ifndef UWB_WLP_CONV_MAJOR
#define UWB_WLP_CONV_MAJOR 0  / * dynamic major by default */
#endif

#ifndef UWB_WLP_CONV_NR_DEVS
#define UWB_WLP_CONV_NR_DEVS 1  / * uwb_wlp_conv0 through uwb_wlp_conv1 */
#endif

int uwb_wlp_conv_major = UWB_WLP_CONV_MAJOR;
int uwb_wlp_conv_minor = 0;
int uwb_wlp_conv_nr_devs = UWB_WLP_CONV_NR_DEVS; / * number of bare scull devices */

module_param(uwb_wlp_conv_major, int, S_IRUGO);
module_param(uwb_wlp_conv_minor, int, S_IRUGO);
module_param(uwb_wlp_conv_nr_devs, int, S_IRUGO);

MODULE_AUTHOR ("Etienne van der Linde");
MODULE_DESCRIPTION("UWB WLP Conv");
MODULE_LICENSE("GPL");

/ * Split minors in two parts */
#define TYPE(minor) (((minor) >> 4) & 0xf) / * high nibble */
#define NUM(minor) ((minor) & 0xf) / * low nibble */

/ * The different configurable parameters */
extern int uwb_wlp_conv_major; / * main.c */
extern int uwb_wlp_conv_nr_devs;

#define NAME_SIZE 30

struct uwb_wlp_conv_defaults{
    char device_name[NAME_SIZE];
};

struct uwb_wlp_conv_defaults DEFAULTS_ARRAY[UWB_WLP_CONV_NR_DEVS] = {
    {"UWB WLP Convergence\0"}
};

struct uwb_wlp_conv_dev {
    uint8_t         local_id;                    / * local identifier */
    char*           device_name;                 / * local identifier */
    uint8_t         status;
    char*           local_data_buffer;           / * Locally stored data */
    size_t          local_data_buffer_size; / * Locally stored data size */
    struct semaphore sem;                        / * mutual exclusion semaphore */
    struct cdev      cdev;                       / * Char device structure */
    struct socket*   ServerSocket;
    struct socket*   TxSocket;
    struct completion texit;
};

#define setup_uwb_wlp_conv_dev(dev,index) \
    dev->local_id = (uint8_t)index; \
    dev->device_name = DEFAULTS_ARRAY[index].device_name;\
    dev->status = 0;\
    dev->local_data_buffer = 0;\
    dev->local_data_buffer_size = 0;\
    dev->ServerSocket = NULL;\
    dev->TxSocket = NULL;

struct uwb_wlp_conv_dev *uwb_wlp_conv_devices; / * allocated in uwb_wlp_conv_init_module */

/ * Prototypes for shared functions */
ssize_t uwb_wlp_conv_read(struct file *filp, char __user *buf, size_t count, loff_t *f_pos);
ssize_t uwb_wlp_conv_write(struct file *filp, const char __user *buf, size_t count, loff_t *f_pos);
loff_t uwb_wlp_conv_llseek(struct file *filp, loff_t off, int whence);
int uwb_wlp_conv_ioctl(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);

/ *****************************************************************************************************
* Debug Printing
****************************************************************************************************/

#define UWB_WLP_CONV_MODULE_DEBUG 1
/ /#define UWB_WLP_CONV_PACKET_DEBUG 1
/ * Module specific debug printing */
#undef MDEBUG / * undef it, just in case */
#ifdef UWB_WLP_CONV_MODULE_DEBUG
# define MDEBUG(fmt, args...) printk(">>uwb_wlp_conv: " fmt, ## args)
#else
# define MDEBUG(fmt, args...) / * not debugging: nothing */
#endif

/ * Packet specific debug printing */
#undef PDEBUG / * undef it, just in case */
#ifdef UWB_WLP_CONV_PACKET_DEBUG
# ifdef __KERNEL__
/ * This one if debugging is on, and kernel space */
/ /# define PDEBUG(fmt, args...) printk( KERN_DEBUG "uwb_wlp_conv: " fmt, ## args)
# define PDEBUG(fmt, args...) printk(">>uwb_wlp_conv: " fmt, ## args)
# else
/ * This one for user space */
# define PDEBUG(fmt, args...) fprintf(stderr, fmt, ## args)
# endif
#else
# define PDEBUG(fmt, args...) / * not debugging: nothing */
#endif

/ *************************************************
* uwb_wlp_conv_print_skb()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct sk_buff *skb_ptr - structure containing the data
*
* return - number of bytes printed
*************************************************/
ssize_t uwb_wlp_conv_print_skb(struct sk_buff *skb_ptr)
{
    int i = 0;
#ifdef UWB_WLP_CONV_PACKET_DEBUG
    for(i=0; i<skb_ptr->len; i++)
    {
            printk("%02X ", (unsigned char)skb_ptr->data[i]);
    }
    printk("\n");
#endif
    return i;
}

/ *****************************************************************************************************
* Functional declarations
****************************************************************************************************/

/ * Thread signals */
DECLARE_WAIT_QUEUE_HEAD(uwb_wlp_conv_wait_threads);

static pid_t        RxPid, TxPid = 0;
static atomic_t     RxThreadActive, TxThreadActive;
static uint8_t      shutdown_request = 0;

#define SERV_PORT 3009
#define CONNECT_TIMEOUT 5000
/ *****************************************************************************************************/

/ * Pointer to upper layer function that handles received data packets */
static int (*uwb_wlp_conv_external_receiver) (struct sk_buff *data_buffer) = NULL;

/ *************************************************
* uwb_wlp_conv_register_receiver_function()
*
* Registers a pointer to an upper layer function to be called
* that should handle packets received from the lower
* level interface.
*
* void* receiver_function - function pointer to register
*
* return - always 0
*************************************************/
int uwb_wlp_conv_register_receiver_function(void* receiver_function)
{
    MDEBUG("Registering receiver function\n");
    uwb_wlp_conv_external_receiver = receiver_function;
    return 0;
}

/ * Export symbol so that external modules can find it */
```

```
EXPORT_SYMBOL(uwb_wlp_conv_register_receiver_function);

/ *****************************************************************************************************
* UWB WLP TX Interface
****************************************************************************************************/
const unsigned char uwb_wlp_default_ip_address[4] = {192,168,0,150};
static char * uwb_wlp_ip_address = "";

static struct socket *writing_socket;

module_param(uwb_wlp_ip_address, charp, 0000);
MODULE_PARM_DESC(uwb_wlp_ip_address, "Destination WLP IP Address");

/ /#define WAIT_FOR_TX_COMPLETE 1

/ **************************************************
* uwb_wlp_conv_send_buffer()
*
* Function to transform data buffer and transmit over the WLP socket
*
* struct socket *sock - WLP socket to use for transmission
* struct sk_buff *buff_data - structure containing the data buffer
*
* return - number of bytes transmitted
**************************************************/
size_t uwb_wlp_conv_send_buffer(struct socket *sock, struct sk_buff *buff_data)
{
    struct msghdr msg;
    mm_segment_t oldfs; / / mm_segment_t is just a long
    struct iovec iov; / / structure containing a base addr. and length
    int return_length;

    PDEBUG("Entering SendBuffer\n");

    / * Setup message red tape */
    msg.msg_name = 0;
    msg.msg_namelen = 0;
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1; / / point to be noted
    msg.msg_control = NULL;
    msg.msg_controllen = 0;
    msg.msg_flags = MSG_NOSIGNAL; / / 0/ *MSG_DONTWAIT*/ ;

    iov.iov_base = (char*) buff_data->data; / / as we know that iovec is
    iov.iov_len = (__kernel_size_t) buff_data->len; / / nothing but a base addr and length

    / / #define get_fs() (current_thread_info()- >addr_limit)
    / / similar for set_fs;

    / * Therefore this line sets the "fs" to KERNEL_DS and saves its old value */
    oldfs = get_fs();
    set_fs(KERNEL_DS);

    / * Actual Sending of the Message */
    return_length = sock_sendmsg(sock, &msg, (size_t)(buff_data->len));

    / / PDEBUG("Sent Buffer of length %d with return %d\n", buff_data->len, return_length);

    / * retrieve the old value of fs (whatever it is) */
    set_fs(oldfs);

    PDEBUG("Leaving SendBuffer\n");
    return return_length;

} « end uwb_wlp_conv_send_buffer »

/ **************************************************
* uwb_wlp_conv_tx_packet_top_entry()
*
* Function to transmit data and then wait for its completion
*
* struct sk_buff *buff_data - structure containing the data buffer
*
* return - number of bytes transmitted
**************************************************/
int uwb_wlp_conv_tx_packet_top_entry(struct sk_buff *buff_data)
{
    int ret = 0;
    if(writing_socket ! = 0)
    {
            PDEBUG("Sending data downwards [%d]: ", buff_data->len);
            uwb_wlp_conv_print_skb(buff_data);
            ret = uwb_wlp_conv_send_buffer(writing_socket, buff_data);
#ifdef WAIT_FOR_TX_COMPLETE
            PDEBUG("Done with uwb_wlp_conv_send_buffer(), waiting for interrupt with queue size %d\n", writing_socket->sk
            ->sk_write_queue.qlen);
            wait_event_interruptible_timeout(*(writing_socket->sk->sk_sleep), skb_queue_empty(&(writing_socket
            ->sk->sk_write_queue)) || shutdown_request ! = 0, HZ/ 100);
#endif
            PDEBUG("Done waiting, exiting uwb_wlp_conv_tx_packet_top_entry() with queue size %d\n", writing_socket->sk
            ->sk_write_queue.qlen);
            return ret;
    }
    else
    {
            PDEBUG("Error! No socket available to send data on! \n");
            return - 1;
    }
} « end uwb_wlp_conv_tx_packet_top_entry »
```

```
/ * Export symbol so that external modules can find it */
EXPORT_SYMBOL(uwb_wlp_conv_tx_packet_top_entry);

/ **************************************************
* uwb_wlp_conv_atoi()
*
* simple function for converting chars to int
*
* const char *name - the victim
*
* return - converted value
**************************************************/
static int uwb_wlp_conv_atoi(const char *name)
{
    int val = 0;
    for (;; name++) {
            switch (*name) {
                case '0'...'9':
                    val = 10*val+(*name- '0');
                break;

                default:
                return val;
            }
    }
}

/ **************************************************
* uwb_wlp_conv_get_addr()
*
* Attempt to obtain a valid WLP IP address from various sources
*
* unsigned char* wlp_ip_address - structure containing the data buffer
*
* return - void
**************************************************/
void uwb_wlp_conv_get_addr(unsigned char* wlp_ip_address)
{
    unsigned char addr_nib[4];
    int i,j,k;
    int err = 0;

    MDEBUG("Obtaining IP address...\n");
    if(uwb_wlp_ip_address)
    {
            MDEBUG("Attempting MODULE PARAM conversion on %s...\n", uwb_wlp_ip_address);

            j = strlen(uwb_wlp_ip_address) - 1;
            if(j<6)
            {
                MDEBUG("Address length = %d too short! \n", j);
                err = 1;
            }
            else
            {
                for(i=3;i>=0;i- -)
                {
                    / /MDEBUG("Converting nibbet %d\n", i);
                    for(k=0;k<2;k++)
                    {
                            addr_nib[k] = '0';
                    }
                    addr_nib[3] = '\0';
                    while((uwb_wlp_ip_address[j] ! = '.') && (k>=0) && (j>=0))
                    {
                            addr_nib[k- -] = uwb_wlp_ip_address[j- - ];
                    }
                    j- -;

                    if((j<0) & (i! =0))
                    {
                        err = 1;
                        MDEBUG("Error in conversion! \n");
                        break;
                    }

                    / /MDEBUG("Convert %s\n",addr_nib);
                    wlp_ip_address[i] = (unsigned char)uwb_wlp_conv_atoi(addr_nib);
                } « end for i=3;i>=0;i- - »
            } « end else »
            if(err) / *An error occurred */
                    memcpy(wlp_ip_address,uwb_wlp_default_ip_address,4);
            else
            {
                / * Check for local address */
                if(((wlp_ip_address[0] == 0) && (wlp_ip_address[1] == 0) && (wlp_ip_address[2] == 0) && (
                wlp_ip_address[3] == 0)) || ((wlp_ip_address[0] == 127) && (wlp_ip_address[1] == 0) && (wlp_ip_address[2] == 0)
                {
                    MDEBUG("Error: address obtained is local address! Using default address...\n");
                    memcpy(wlp_ip_address,uwb_wlp_default_ip_address,4);
                }
            }
    } « end if uwb_wlp_ip_address »
    else
            memcpy(wlp_ip_address,uwb_wlp_default_ip_address,4);
    return;
} « end uwb_wlp_conv_get_addr »

/ **************************************************
* uwb_wlp_conv_close_socket()
```

```
*
* Special function for closing the WLP socket
*
* struct socket *sock2close - WLP socket to close
*
* return - void
**************************************************/
void uwb_wlp_conv_close_socket(struct socket *sock2close)
{
    struct socket *sock;

    if( sock2close== NULL )
            return;
    MDEBUG("Closing socket\n");

    sock = sock2close;
    sock2close = NULL;
    sock_release(sock);
}

/ **************************************************
* uwb_wlp_conv_set_up_client_socket()
*
* Open a socket on the UWB WLP intantiated interface
*
* unsigned int IP_addr - WLP IP address to use
* int port_no - Port number to use over interface
*
* return - WLP socket
**************************************************/
struct socket * uwb_wlp_conv_set_up_client_socket(unsigned int IP_addr, int port_no)
{
    struct socket *clientsock;
    struct sockaddr_in sin;
    int error, i;

    / * First create a socket */
    error = sock_create(PF_INET,SOCK_STREAM,IPPROTO_TCP,&clientsock);
    if (error<0)
    {
            MDEBUG("Error during creation of socket; terminating\n");
            return 0;
    }

    / * Now bind and connect the socket */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = htonl(IP_addr);
    sin.sin_port = htons(port_no);
    for(i=0; i<10; i++)
    {
            error = clientsock->ops- >connect(clientsock,(struct sockaddr*)&sin,sizeof(sin),0);
            if (error<0)
            {
                MDEBUG("Error connecting client socket to server: %i, retrying .. %d \n",error, i);
                if(i == 10- 1)
                {
                    MDEBUG("Giving Up! \n");
                    return 0;
                }
                interruptible_sleep_on_timeout(clientsock->sk->sk_sleep,(CONNECT_TIMEOUT*HZ)/ 1000);
            }
            else
                break; / / connected
    }
    return clientsock;
} « end uwb_wlp_conv_set_up_client_socket »

/ **************************************************
* uwb_wlp_conv_setup_tx_connection_thread()
*
* Manages the setup of a WLP socket for transmitting data.
*
* void *info - Info passed from instantiator - ignored here
*
* return - always 0
**************************************************/
int uwb_wlp_conv_setup_tx_connection_thread(void *info)
{
    unsigned char destination_IP[4];
    unsigned long destination_address = 0;

    atomic_inc(&TxThreadActive);

    daemonize("uwb_wlp_conv_setup_tx_connection_thread");
    allow_signal(SIGTERM);

    MDEBUG("uwb_wlp_conv_setup_tx_connection thread started...\n");

    uwb_wlp_conv_get_addr(destination_IP);
    (void) memcpy(&destination_address, destination_IP, 4);
    MDEBUG("Attempting to establish an outgoing connection to %d.%d.%d.%d...\n", destination_IP[0],destination_IP[1], destination_IP[2],destination_IP[3]);

    writing_socket = uwb_wlp_conv_set_up_client_socket(ntohl(destination_address), SERV_PORT);

    if(writing_socket > 0)
    {
            MDEBUG("Outgoing connection setup successfull\n");
            printk("WLP TX connection established.\n");
    }

    MDEBUG("Exiting uwb_wlp_conv_setup_tx_connection_thread...\n");
    atomic_dec(&TxThreadActive);
    wake_up(&uwb_wlp_conv_wait_threads);

    return 0;
} « end uwb_wlp_conv_setup_tx_connection_thread »


/ *************************************************************************************************
* UWB WLP RX Interface
*************************************************************************************************/
struct reading_thread_data {
    struct uwb_wlp_conv_dev       *parent_dev;
    struct socket                 * newsock;
    pid_t                         processID;
    atomic_t                      DaemonCount;
    struct reading_thread_data    *next_in_list;
};

static struct reading_thread_data* rtd_list_head;
spinlock_t rx_info;

#define MAX_RECEIVE_BUFFER_SIZE 1600
#define THREAD_KILL_ATTEMPTS 10

/ **************************************************
* uwb_wlp_conv_rx_packet_top_exit()
*
* Calls the externel upperlayer function, if it was registered,
* for handling the data packet
*
* struct sk_buff *buff_data - structure containing the data buffer
*
* return - 0 on success, - 1 on failure
**************************************************/
int uwb_wlp_conv_rx_packet_top_exit(struct sk_buff *rx_buff)
{
    PDEBUG("Sending data upwards [%d]: ", rx_buff->len);
    uwb_wlp_conv_print_skb(rx_buff);

    if(uwb_wlp_conv_external_receiver == NULL)
    {
            PDEBUG("Error! No receiver function registered to service received packet! \n");
            return - 1;
    }
    else
    {
            PDEBUG("Calling external receiver function! \n");
            uwb_wlp_conv_external_receiver(rx_buff);
            return 0;
    }
}

/ **************************************************
* uwb_wlp_conv_receive_buffer()
*
* Recieves data on the socket and puts it in the data buffer
*
* struct socket *sock - WLP socket to use for transmission
* struct sk_buff *buff_data - structure containing the data buffer
*
* return - length of data recieved
**************************************************/
size_t uwb_wlp_conv_receive_buffer(struct socket *sock, struct sk_buff *buff_data)
{
    struct msghdr msg;
    struct iovec iov;
    int len = buff_data->truesize - sizeof(struct sk_buff) - 1;
    mm_segment_t oldfs;

    / * Set the msghdr structure*/
    msg.msg_name = 0;
    msg.msg_namelen = 0;
    msg.msg_iov = &iov;
    msg.msg_iovlen = 1;
    msg.msg_control = NULL;
    msg.msg_controllen = 0;
    msg.msg_flags = 0;

    / * Set the iovec structure*/
    iov.iov_base = (void *) buff_data->data;
    iov.iov_len = (size_t) len;

    / * Recieve the message */
    oldfs = get_fs();
    set_fs(KERNEL_DS);

    len = sock_recvmsg(sock,&msg,len,0/ *MSG_DONTWAIT*/ ); / / let it wait if there is no message
    / / len = kernel_recvmsg(sock,&msg,Length,MSG_WAITALL); / / let it wait if there is no message

    set_fs(oldfs);

    / / if ((len! =- EAGAIN)&&(len! =0))
    / / PDEBUG("RecvBuffer Recieved %i bytes \n",len);

    / * Add socket to sk_buff */
    buff_data->sk = sock->sk;
    buff_data->next = 0;

    return len;
} « end uwb_wlp_conv_receive_buffer »
```

```
/ *************************************************
* uwb_wlp_conv_rx_thread()
*
* Thread waiting for data to be received on socket, and then
* passing it up for processing
*
* void *info - Info passed from instantiator
*
* return - always 0
*************************************************/
int uwb_wlp_conv_rx_thread(void *info)
{
   struct reading_thread_data* passed_data = (struct reading_thread_data *) info;
   struct socket* reading_socket = passed_data->newsock;
   / / char rcv_buffer[MAX_RECEIVE_BUFFER_SIZE] = "";
   / / struct sk_buff *rcv_buffer = alloc_skb(MAX_RECEIVE_BUFFER_SIZE, GFP_ATOMIC);
   struct sk_buff *rcv_buffer = 0;

   / * Thread red tape */
   atomic_inc(&(passed_data->DaemonCount)); / / keep track of what is running
   daemonize("uwb_wlp_conv_rx_thread"); / / disable all signals
   allow_signal(SIGTERM); / / except this one

   MDEBUG("Reading thread started on PID %d with DC %d...\n", passed_data->processID, atomic_read(&(passed_data->
   DaemonCount)));
   printk("UWB WLP CONV RX connection established! \n");

   / * Kill the cloned server socket ?*/
   / / uwb_wlp_conv_close_socket(dev->ServerSocket);
   / * The constant running part of the thread */
   / *************************************/

   if(reading_socket ! = 0)
   {
         / * Block all signals except SIGKILL, SIGTERM */
         recalc_sigpending();
         while(! signal_pending(current) && shutdown_request == 0)
         {
            rcv_buffer = alloc_skb(MAX_RECEIVE_BUFFER_SIZE, GFP_ATOMIC);

            rcv_buffer->len= uwb_wlp_conv_receive_buffer(reading_socket, rcv_buffer);
            if(((signed int)rcv_buffer->len) > 0)
            {
               PDEBUG("Passing data upwards for signalling....\n");
               uwb_wlp_conv_rx_packet_top_exit(rcv_buffer);
            }
            else
            {
               MDEBUG("Error reading received data: %d! \n", rcv_buffer->len);
            }

            wait_event_interruptible(*(reading_socket->sk->sk_sleep), ! skb_queue_empty(&(reading_socket->sk->sk_receive_queue)) ||
            shutdown_request ! = 0);

            PDEBUG("Done waiting for an event. Queue size is %d\n", reading_socket->sk->sk_receive_queue.qlen);

            if(rcv_buffer) kfree_skb(rcv_buffer);
         } « end while ! signal_pending(curre... »
   } « end if reading_socket! =0 »

   / *************************************/
   uwb_wlp_conv_close_socket(reading_socket);
   if(rcv_buffer)
         kfree_skb(rcv_buffer);

   atomic_dec(&(passed_data->DaemonCount));
   wake_up(&uwb_wlp_conv_wait_threads);

   MDEBUG("Exiting reading thread PID %d with DC %d...\n", passed_data->processID, atomic_read(&(passed_data->DaemonCount)));
   return 0;
} « end uwb_wlp_conv_rx_thread »

/ *************************************************
* uwb_wlp_conv_server_accept_connection()
*
* Accepts a new connection on a listening socket
* 1. Create a new socket
* 2. Call socket->ops- >accept
* 3. return the newly created socket
*
* struct socket *sock - Socket to listen on
*
* return - New spawned WLP socket
*************************************************/
struct socket* uwb_wlp_conv_server_accept_connection(struct socket *sock)
{
   struct socket * newsock;
   int error;

   if (sock==NULL) return 0;

   MDEBUG("Duplicate socket\n");

   / * Before accept: Clone the socket */
   error = sock_create(PF_INET,SOCK_STREAM,IPPROTO_TCP,&newsock);
   if (error<0)
         MDEBUG("Error during creation of the other socket; terminating\n");

   newsock->type = sock->type;
   newsock->ops=sock->ops;
```

```
    newsock->sk->sk_reuse = 1;

    MDEBUG("Waiting to accept on...\n");

    / * Do the actual accept */
    error = newsock->ops- >accept(sock,newsock,0);
    if (error<0)
    {
            / * must add release the socket code here */
            MDEBUG("Error accepting socket\n") ;
            return 0;
    }
    else MDEBUG("New socket accepted ...\n");

    MDEBUG("Leaving server_accept_connection()\n");
    return newsock;
} « end uwb_wlp_conv_server_accept_connection »

/ *************************************************
* uwb_wlp_conv_setup_server_socket()
*
* Open a server socket on the UWB WLP intantiated interface
* 1. Create a new socket
* 2. Bind the address to the socket
* 3. Start listening on the socket
*
* int port_no - Port number to use over interface
*
* return - WLP socket
*************************************************/
struct socket* uwb_wlp_conv_setup_server_socket(int port_no)
{
    struct socket *sock;
    struct sockaddr_in sin;
    int error;

    / * First create a socket */
    error = sock_create(PF_INET,SOCK_STREAM,IPPROTO_TCP,&sock) ;
    if (error<0)
            MDEBUG("Error during creation of socket; terminating\n");

    / * Now bind the socket */
    sin.sin_family = AF_INET;
    sin.sin_addr.s_addr = INADDR_ANY;
    sin.sin_port = htons(port_no);

    error = sock->ops->bind(sock,(struct sockaddr*)&sin,sizeof(sin));
    if (error<0)
    {
            MDEBUG("Error binding socket \n");
            return 0;
    }
    / * Now, start listening on the socket */
    error=sock->ops- >listen(sock,32);
    if (error! =0)
            MDEBUG("Error listening on socket \n");

    / * Now start accepting */
    / / Accepting is performed by the function server_accept_connection

    sock->sk->sk_reuse = 1;

    MDEBUG("Listening on socket ...\n");

    return sock;
} « end uwb_wlp_conv_setup_server_socket »

/ *************************************************
* uwb_wlp_conv_wait4connection_thread()
*
* Thread waiting for a connection to be made on the socket,
* and then spawning a child socket for it
*
* void *info - Info passed from instantiator
*
* return - always 0
*************************************************/
int uwb_wlp_conv_wait4connection_thread(void *info)
{
    struct uwb_wlp_conv_dev *dev = (struct uwb_wlp_conv_dev *) info;
    struct reading_thread_data *current_list_item, *new_list_item;

    atomic_inc(&RxThreadActive);
    daemonize("uwb_wlp_conv_wait4connection");
    allow_signal(SIGTERM);

    MDEBUG("Connection waiting thread started...\n");
    current_list_item = kmalloc(sizeof(struct reading_thread_data), GFP_KERNEL);
    current_list_item->next_in_list = NULL;
    rtd_list_head = current_list_item;

    / * The constant running part of the thread */
    / *************************************/

    while(! signal_pending(current) && shutdown_request == 0)
    {
            current_list_item->newsock = uwb_wlp_conv_server_accept_connection(dev->ServerSocket);
            current_list_item->DaemonCount.counter = 0;
            if(current_list_item->newsock > 0)
            {
                MDEBUG("New connection accepted! Forking a child thread for it...\n");
                / * Fork a child thread for reading on this socket */
                new_list_item = kmalloc(sizeof(struct reading_thread_data), GFP_KERNEL);
                current_list_item->parent_dev = dev;
                current_list_item->next_in_list = new_list_item;
                current_list_item->processID = kernel_thread(uwb_wlp_conv_rx_thread, (void *)(current_list_item) ,0 );

                / * Kill the cloned server socket ?*/
                / / CloseSocket(current_list_item->newsock);

                / * Move the list on */
                current_list_item = new_list_item;
                current_list_item->next_in_list = NULL;
            }
    } « end while ! signal_pending(curre... »
    / ***************************************/
    MDEBUG("Exiting connection waiting thread...\n");

    / * Stop the server listening socket */
    uwb_wlp_conv_close_socket(dev->ServerSocket);

    atomic_dec(&RxThreadActive);
    wake_up(&uwb_wlp_conv_wait_threads);
    complete_and_exit(&dev->texit, 0);
    return 0;
} « end uwb_wlp_conv_wait4connection_thread »

/ ***************************************************************************************************
* Module Driver Functions
****************************************************************************************************/

/ *************************************************
* uwb_wlp_conv_exit()
*
* Standard operations to remove device driver from system.
* Also send signals to kill all the running threads, and then wait
* for them to finish up.
* void
*
* return - void
**************************************************/
static void __exit uwb_wlp_conv_exit(void)
{
    dev_t devno = MKDEV(uwb_wlp_conv_major, uwb_wlp_conv_minor);
    struct reading_thread_data *rtd_list_item, *store_item;
    int timeout_count = THREAD_KILL_ATTEMPTS;

    MDEBUG("Exiting UWB WLP Conv Module %s\n", UWB_WLP_CONV_DRIVER_VERSION);
    / *****************************************************************/

    shutdown_request = 1;

    / * Kill the reading threads */
    if(rtd_list_head)
    {
            rtd_list_item = rtd_list_head;
            while(rtd_list_item->next_in_list ! = NULL)
            {
                MDEBUG("Attempting to kill thread PID %d with DC %d...\n", rtd_list_item->processID, atomic_read(&(
                rtd_list_item->DaemonCount)));
                while(atomic_read(&(rtd_list_item->DaemonCount)) && timeout_count)
                {
                    if(kill_proc(rtd_list_item->processID, SIGTERM, 1) < 0)
                    {
                        atomic_dec(&(rtd_list_item->DaemonCount));
                    }
                    else
                    {
                        MDEBUG("Waiting on thread PID %d with DC %d to be killed...\n", rtd_list_item->processID,
                        atomic_read(&(rtd_list_item->DaemonCount)));
                        wait_event_interruptible_timeout(uwb_wlp_conv_wait_threads, ! atomic_read(&(rtd_list_item->
                        DaemonCount)), HZ);
                    }
                    timeout_count- -;
                    if(! atomic_read(&(rtd_list_item->DaemonCount)))
                        MDEBUG("Thread PID %d killed with DC now %d...\n", rtd_list_item->processID, atomic_read(&(rtd_list_item->DaemonCount)));
                }
                store_item = rtd_list_item;
                rtd_list_item = rtd_list_item->next_in_list;
                kfree(store_item);
            } « end while rtd_list_item->next_i... »
    } « end if rtd_list_head »

    if(atomic_read(&(RxThreadActive)))
    {
            if(kill_proc(RxPid, SIGTERM, 1) < 0)
            {
                /* Something weird probably happened to the main daemon ! ! !
                * Don't wait for it to stop */
                atomic_dec(&(RxThreadActive));
            }
            else wait_event_interruptible(uwb_wlp_conv_wait_threads, ! atomic_read(&(RxThreadActive)));
            MDEBUG("RxPid killed\n");
    }

    if(atomic_read(&(TxThreadActive)))
    {
            if(kill_proc(TxPid, SIGTERM, 1) < 0)
```

```
            {
                atomic_dec(&(TxThreadActive));
            }
            else wait_event_interruptible(uwb_wlp_conv_wait_threads, ! atomic_read(&(TxThreadActive)));
    }
    MDEBUG("TxPid killed\n");
    / * Stop the outgoing socket */
    if(writing_socket)
    {
    uwb_wlp_conv_close_socket(writing_socket);
    }
    / *******************************************************************/

    MDEBUG("Freeing devices\n");
    / * Get rid of our char dev entries */
    if (uwb_wlp_conv_devices) {
            kfree(uwb_wlp_conv_devices);
    }

    MDEBUG("Unregistering\n");
    / * cleanup_module is never called if registering failed */
    unregister_chrdev_region(devno, uwb_wlp_conv_nr_devs);
} « end uwb_wlp_conv_exit »

/ **************************************************
* uwb_wlp_conv_init()
*
* Standard operations to setup and add device driver to system.
* Setup TX and RX threads.
*
* void
*
* return - 0 if success
**************************************************/
static int __init uwb_wlp_conv_init(void)
{
    int result;
    dev_t dev = 0; / /MKDEV(UWB_WLP_CONV_MAJOR, 0);

    MDEBUG("Initializing UWB WLP Conv Module %s...\n", UWB_WLP_CONV_DRIVER_VERSION);

    / * Get a range of minor numbers to work with, asking for a dynamic major unless directed otherwise at load time. */
    if (uwb_wlp_conv_major) {
            dev = MKDEV(uwb_wlp_conv_major, uwb_wlp_conv_minor);
            result = register_chrdev_region(dev, uwb_wlp_conv_nr_devs, "uwb_wlp_conv");
    } else {
            MDEBUG("Atempting device allocation...\n");
            result = alloc_chrdev_region(&dev, uwb_wlp_conv_minor, uwb_wlp_conv_nr_devs, "uwb_wlp_conv");
            uwb_wlp_conv_major = MAJOR(dev);
            MDEBUG("..done! \n");
    }
    if (result < 0) {
            printk(KERN_WARNING "uwb_wlp_conv: can't get major %d\n", uwb_wlp_conv_major);
            MDEBUG("can't get major %d\n", uwb_wlp_conv_major);
            return result;
    }
    MDEBUG("Module registered with major device number %d\n", uwb_wlp_conv_major);

    / * allocate the device - - we can't have them static, as the number can be specified at load time */
    uwb_wlp_conv_devices = kmalloc(uwb_wlp_conv_nr_devs * sizeof(struct uwb_wlp_conv_dev), GFP_KERNEL);
    if (! uwb_wlp_conv_devices) {
            result = -ENOMEM;
            goto fail; / * Make this more graceful */
    }
    memset(uwb_wlp_conv_devices, 0, uwb_wlp_conv_nr_devs * sizeof(struct uwb_wlp_conv_dev));

    / *******************************************************************/
    / * Setup the TX and RX threads */

    / * initialize atomic counters */
    RxThreadActive.counter = 0;
    TxThreadActive.counter = 0;

    spin_lock_init(&rx_info);

    / * Setup a listening server socket */
    uwb_wlp_conv_devices[0].ServerSocket = uwb_wlp_conv_setup_server_socket(SERV_PORT);

    / * Initialize new listening thread */
    init_completion(&(uwb_wlp_conv_devices[0].texit));
    RxPid = kernel_thread(uwb_wlp_conv_wait4connection_thread, (void *)(&(uwb_wlp_conv_devices[0])) ,0 / * no clone flags */ );

    / * Setup an outgoing writing socket */
    TxPid = kernel_thread(uwb_wlp_conv_setup_tx_connection_thread, (void *)(&(uwb_wlp_conv_devices[0])) ,0 / * no clone flags */ );
    / *******************************************************************/
    return 0; / * succeed */

    fail:
            uwb_wlp_conv_exit();

    return result;
} « end uwb_wlp_conv_init »

module_init(uwb_wlp_conv_init);
module_exit(uwb_wlp_conv_exit);
```

D.3 UWB Link Control User Application

This section lists the source code of the UWB Link Control user application described in section 3.2.5.3.

D.3.1 UWB Link Control User Application Source Files

```
/*****************************************************************
* UWB Link Control User Application
* As part of the Bluetooth over Ultra Wideband Evaluation Research Project
*
* Copyright (C) University of Pretoria
* Etienne van der Linde <etienne.vdlinde@gmail.com>
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License version
* 2 as published by the Free Software Foundation.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
* MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
* GNU General Public License for more details.
*
*****************************************************************
*
* This user application provides a command line interface to control the
* data routing performed by the developed UWB Router Kernel Module.
* Different insertion points as source of the (Bluetooth generated) data
* stream can be selected, and different lower level targets (with a focus
* the WLP UWB interface) can be selected as destination for the routed
* data packets. The program usage is a follows:
*
*   uwb_link_control <MODE>
*
*   Modes:
*
*   t       Set insertion point to UWB Controller Test Data
*   c       Set insertion point to L2CAP Insertion Point
*   h       Set insertion point to HCI Insertion Point
*
*   n       Set top routing to No Routing Mode
*   p       Set top routing to Straight Ethernet Passthrough Mode
*   u       Set top routing to UWB Hardware Mode
*   l       Set top routing to Local Loopback Mode
*
*   q       Query current routing mode
*   x       Exit
*
*   Data generation:
*   0 - 9   Send data set [0-9]
*
*****************************************************************/
#include <unistd.h>
#include <fcntl.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <ctype.h>
#include "../uwb_router/uwb_router_src.h"

#define BUFSZ 32
#define UWB_LINK_CONTROL_DEBUG 1

static char read_buffer[BUFSZ];
static int rfd = 0;

/* Macros to help debugging */
#undef PDEBUG /* undef it, just in case */
#ifdef UWB_LINK_CONTROL_DEBUG
#define PDEBUG(fmt, args...) printf(fmt, ## args)
#else
#define PDEBUG(fmt, args...) /* not debugging: nothing */
#endif

/**************************************************
* print_buffer()
*
* Displays the contents of a buffer for debugging purpouses
*
* char* buffer - pointer to buffer containing the data
* int size - size of the buffer
*
* return - void
**************************************************/
void print_buffer(char* buffer, int size)
{
   int i;
   for(i=0; i<size; i++)
        printf("%02X ", (unsigned char)buffer[i]);
   printf("\n");
}
```

```
/ **************************************************
* gen_buffer()
*
* Generates test data to exercise the link
*
* char* buffer - pointer to buffer that should contain the data
* int size - size of the buffer
*
* return - size of the buffer
**************************************************/
ssize_t gen_buffer(char* buffer, int size)
{
    int seed;
    for(seed=0; seed<size; seed++)
            buffer[seed] = (unsigned char)(((0xFA45 - seed) + (seed << 11)) * ((0x1234 + seed) - (seed << 6))) ^ ((0x7A51 + seed) + (seed << 2));
    return seed;
}

/ **************************************************
* write_buffer()
*
* Send data buffer down to driver for writing
*
* int device - device identifier
* char* buffer - buffer containing the data
* int size - number of bytes to be written
*
* return - number of bytes written successfully
**************************************************/
int write_buffer(int device, char* buffer, int size)
{
    int retval = 0;
    printf("Writing buffer: ");
    print_buffer(buffer, size);

    / * Write data */
    retval = write(device, buffer, size);

    PDEBUG("Done with retval %d\n", retval);
    return retval;
}

/ **************************************************
* sighandler()
*
* Function for handling asyncronous notifications from the driver
* Reads data received and prints it
*
* int signo - the signal to react uppon or not
*
* return - void
**************************************************/
void sighandler(int signo)
{
    int length = BUFSZ;

    if (signo==SIGIO)
    {
            PDEBUG("Asynchronous notification received! \n");
            while(length == BUFSZ)
            {
                length = read(rfd, read_buffer, BUFSZ);

                if(length <= 0)
                    printf("Error receiving data! \n");
                else
                {
                    printf("Received buffer [%d]: ", length);
                    print_buffer(read_buffer, length);
                }
                memset(read_buffer, 0, BUFSZ);
            }
    }
    return;
} « end sighandler »

/ **************************************************
* print_usage()
*
* Displays information on how to use this program
*
* return - void
**************************************************/
void print_usage(void)
{
    int i;

    printf("\nUsage: uwb_link_control MODE\n\n");
    printf("Modes:\n");
    for(i=0;i<INS_MODE_END;i++)
    {
            printf("\t%c\tSet insertion point to %s\n", uwb_router_ins_mode_description[i].identifier,
            uwb_router_ins_mode_description[i].description);
    }
    printf("\n");
    for(i=0;i<ROUTING_MODE_END;i++)
    {
            printf("\t%c\tSet top routing to %s\n", uwb_router_routing_mode_description[i].identifier,
            uwb_router_routing_mode_description[i].description);
    }
    printf("\n");
    printf("\n");
    printf("\tq\tQuery current routing mode\n\n");
    printf("\tx\tExit\n\n");
    printf("Data generation:\n");
    printf("\t0 - 9\tSend data set [0- 9]\n\n");
} « end print_usage »

/ **************************************************
* simple_strtoul()
*
* Conversion of commandline hex values in string format to unsigned long
*
* const char *cp - pointer to string to be converted
* char **endp - pointer to end of string
* unsigned int base - 16 for hex, 10 for natural
*
* return - the converted value
*************************************************/
unsigned long simple_strtoul(const char *cp,char **endp,unsigned int base)
{
    unsigned long result = 0,value;
    if (! base) {
            base = 10;
            if (*cp == '0') {
                base = 8;
                cp++;
                if ((toupper(*cp) == 'X') && isxdigit(cp[1])) {
                    cp++;
                    base = 16;
                }
            }
    } else if (base == 16) {
            if (cp[0] == '0' && toupper(cp[1]) == 'X')
            cp += 2;
    }
    while (isxdigit(*cp) &&
    (value = isdigit(*cp) ? *cp- '0' : toupper(*cp)- 'A'+10) < base) {
            result = result*base + value;
            cp++;
    }
    if (endp)
            *endp = (char *)cp;
    return result;
} « end simple_strtoul »

/ **************************************************
* print_mode()
*
* Displays the current insertion and routing mode of the system
*
* int routing_mode - mode the routing is set to
* int ins_mode - mode insertion is set to
*
* return - void
*************************************************/
void print_mode(int routing_mode, int ins_mode)
{
    printf("Insertion mode is set to: %s\n", uwb_router_ins_mode_description[ins_mode].description);
    printf("Routing mode is set to: %s\n", uwb_router_routing_mode_description[routing_mode].description);
}

/ **************************************************
* main()
*
* Main Entry point
* Open handle to device
* Setup asynchronous signal handling from device
* Parse input to
* - Setup device routing and insertion mode
* - Generate and send test data on demand
*
* int argc - number of commandline argurments
* char* argv - the commandline argurments
*
* return - 0 on success
*************************************************/
int main(int argc, char* argv[])
{
    int fd;
    char i;
    unsigned short FirmwareVersion;
    char command_claimed = 0;
    char identifier = UWB_LINK_CONTROL_MAGIC;
    unsigned short read_val = 0;
    int return_val = 0;
    char device[] = "/ dev/ uwb_router\0";
    char mode = 'Q';
    int driver_version = 0;
    char current_mode = 0, routing_mode = 0, ins_mode = 0;
    char writing_buffer[BUFSZ];
    int write_length = 0;
    struct sigaction action;

    / * Retrieve the requested mode from the commandline */
    if(argc > 1)
    {
            mode = toupper(argv[1][0]);
    }

    / * Open device */
    printf("Opening device %s\n", device);
    fd = open(device, O_RDWR); / / | O_NONBLOCK);
    if (fd < 0) {
            printf("Error opening device %s - code %d given! \n",device, fd);
            exit(1);
    }

    / * Check driver version */
    ioctl(fd, UWB_ROUTER_GET_DRIVER_VERSION, (int*) &driver_version);
    printf("Driver version is: %d\n", driver_version);

    / * Report to driver that it is indeed us on this side */
    ioctl(fd, UWB_ROUTER_REPORT_LINK_CONTROL_PRESENCE, (char*) &identifier);

    / * Cleanup the reading buffer */
    memset(read_buffer, 0, BUFSZ);
    rfd = fd;

    / * Initialize asynchronous notifier for receive and sighandler() for reaction */
    memset(&action, 0, sizeof(action));
    action.sa_handler = sighandler;
    action.sa_flags = 0;
    sigaction(SIGIO, &action, NULL);
    fcntl(fd, F_SETOWN, getpid());
    fcntl(fd, F_SETFL, fcntl(fd, F_GETFL) | FASYNC);

    / * Check and display the current routing mode */
    routing_mode = (char)ioctl(fd, UWB_ROUTER_GET_ROUTING_MODE);
    ins_mode = (char)ioctl(fd, UWB_ROUTER_GET_INS_MODE);
    print_mode(routing_mode, ins_mode);

    / * Setup commandline control */
    printf("Enter your command...\n");
    while('X' ! = (mode = toupper(getc(stdin))))
    {
            command_claimed = 0;
            switch(mode)
            {
                    case 'Q':
                        printf("Querying device for current routing mode...\n");
                        routing_mode = (char)ioctl(fd, UWB_ROUTER_GET_ROUTING_MODE);
                        ins_mode = (char)ioctl(fd, UWB_ROUTER_GET_INS_MODE);
                        print_mode(routing_mode, ins_mode);
                        command_claimed = 1;
                    continue;

                    case 0x0A:
                        command_claimed = 1;
                    continue; / * Don't respond on the enter key */
            }

            / * Check for insertion mode */
            for(i=0;i<INS_MODE_END;i++)
            {
                    if(toupper(uwb_router_ins_mode_description[i].identifier) == mode)
                    {
                            printf("Setting insertion mode to %s...\n", uwb_router_ins_mode_description[i].description);
                            ioctl(fd, UWB_ROUTER_SET_INS_MODE, (char*)&i);
                            command_claimed = 1;
                            break;
                    }
            }

            / * Check for top routing mode */
            for(i=0;i<ROUTING_MODE_END;i++)
            {
                    if(toupper(uwb_router_routing_mode_description[i].identifier) == mode)
                    {
                        printf("Setting routing mode to %s...\n", uwb_router_routing_mode_description[i].description);
                        ioctl(fd, UWB_ROUTER_SET_ROUTING_MODE, (char*)&i);
                        command_claimed = 1;
                        break;
                    }
            }

            / * Generate and send test data of various sizes to exercise the link */
            if(('0' <= mode) & (mode <= '9'))
            {
                    printf("Sending data set %c...\n", mode);
                    write_length = (int)(mode - '0');
                    if(write_length == 0)
                            write_length = 10;
                    gen_buffer(writing_buffer, write_length);
                    write_buffer(fd, writing_buffer, write_length);
                    command_claimed = 1;
            }

            / * no command claimed yet */
            if(command_claimed == 0) print_usage();
    } « end while 'X'! =(mode=toupper(ge... »

    / * Remove presence from driver */
    identifier = 0;
    ioctl(fd, UWB_ROUTER_REPORT_LINK_CONTROL_PRESENCE, (char*) &identifier);

    close(fd);

    return 0;
} « end main »
```

D.4 Bluetooth Stack Modifications

This section lists the modifications performed to the BlueZ Bluetooth stack as described in section 3.2.5.1.

D.4.1BlueZ l2cap.c modifications

```
#define UWB_ROUTER_CONNECT_TO_BLUEZ 1

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
/****************************************************************************************
* BToUWB Insertion
****************************************************************************************/
#include "../../uwb_router/uwb_router_src.h"

#define L2CAP_OVER_UWB_STACK_DEBUG 1
//#define L2CAP_OVER_UWB_PACKET_DEBUG 1

#undef BDEBUG /* undef it, just in case */
#ifdef L2CAP_OVER_UWB_STACK_DEBUG
#define BDEBUG(fmt, args...) printk(KERN_ERR "L2CAP> "); printk(KERN_ERR fmt, ## args)
#else
#define BDEBUG(fmt, args...)
#endif

#undef PDEBUG /* undef it, just in case */
#ifdef L2CAP_OVER_UWB_PACKET_DEBUG
#define PDEBUG(fmt, args...) printk(KERN_ERR "L2CAP> "); printk(KERN_ERR fmt, ## args)
#else
#define PDEBUG(fmt, args...)
#endif

/***************************************************
* l2cap_over_uwb_print_skb()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct sk_buff *skb_ptr - structure containing the data
*
* return - number of bytes printed
***************************************************/
ssize_t l2cap_over_uwb_print_skb(struct sk_buff *skb_ptr)
{
    int i = 0;
#ifdef L2CAP_OVER_UWB_PACKET_DEBUG
    for(i=0; i<skb_ptr->len; i++)
    {
        printk(KERN_ERR "%02X ", (unsigned char)skb_ptr->data[i]);
    }
    printk(KERN_ERR "\n");
#endif
    return i;
}
static char l2cap_over_uwb_routing_mode = 0;

/***************************************************
* l2cap_over_uwb_set_routing_mode()
*
* Function for enabling / disabling the routing of packets over
* the UWB Routing subsystem
*
* char mode - signifies the requested mode
*
* return - void
***************************************************/
void l2cap_over_uwb_set_routing_mode(char mode)
{
    BDEBUG("L2CAP setting routing mode to %d\n", mode);
    l2cap_over_uwb_routing_mode = mode;
}

EXPORT_SYMBOL(l2cap_over_uwb_set_routing_mode);

/* Pointer to lower layer function that handles data packets to be transmitted */
static int (*l2cap_over_uwb_external_transmitter) (struct sk_buff *data, char mode) = NULL;

/***************************************************
* l2cap_over_uwb_register_transmitter_function()
*
* Registers a pointer to a lower layer function to be called
* that should handle packets transmitted from the upper
* level Bluetooth stack.
*
* void* transmitter_function - function pointer to register
*
* return - always 0
***************************************************/
int l2cap_over_uwb_register_transmitter_function(void* transmitter_function)
{
    BDEBUG("Registering transmitter function\n");
    l2cap_over_uwb_external_transmitter = transmitter_function;
    return 0;
}
```

```
/ * Export symbol so that external modules can find it */
EXPORT_SYMBOL(l2cap_over_uwb_register_transmitter_function);

#define L2CAP_OVER_UWB_ID 'H'
#define L2CAP_OVER_UWB_HEADER_SIZE 16

struct l2cap_over_uwb_header
{
   __u8 id;
   __u16 len;
   bdaddr_t src;
   bdaddr_t dst;
   __u8 type;
}__attribute__ ((packed));

/ **************************************************
* l2cap_over_uwb_extract_connection_info()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct hci_conn *conn - structure containing connection info
* struct sk_buff *in - data packet with connection info added
* struct sk_buff *out - data packet with connection info removed
*
* return - 0 on success, else negative
**************************************************/
int l2cap_over_uwb_extract_connection_info(struct hci_conn **conn, struct sk_buff *in, struct sk_buff *out)
{
   struct l2cap_over_uwb_header *hdr = (struct l2cap_over_uwb_header *) in->data;
   struct hci_dev *hdev = 0;
   struct l2cap_conn *new_l2cap_conn = 0;
   int new_length;

   / * Check ID */
   if(hdr->id != L2CAP_OVER_UWB_ID)
   {
         BDEBUG("Unexpected ID in header! \n");
         return - 1;
   }

   new_length = (int)__le16_to_cpu(hdr->len);

   if(new_length ! = (in->len - L2CAP_OVER_UWB_HEADER_SIZE))
   {
         BDEBUG("Unexpected length %d in header, needed %d! \n", new_length, (in->len -
         L2CAP_OVER_UWB_HEADER_SIZE));
         return - 2;
   }

   hdev = hci_get_route(&(hdr->src), &(hdr->dst));
   if(! hdev)
   {
         BDEBUG("Could not allocate hdev! \n");
         return - 3;
   }

   *conn = hci_conn_hash_lookup_ba(hdev, 0x01, &(hdr->src));
   if(! *conn)
   {
         BDEBUG("Could not allocate hci_con! \n");
         return - 4;
   }

   new_l2cap_conn = l2cap_conn_add(*conn, 0);
   if(! new_l2cap_conn)
   {
         BDEBUG("Could not allocate l2cap_con! \n");
         return - 5;
   }

   if (in->len) {
         memcpy(skb_put(out, new_length), &(in->data[L2CAP_OVER_UWB_HEADER_SIZE]), new_length);
   }

   out->dev = (void *) hdev;

   bt_cb(out)->pkt_type = hdr->type;
   bt_cb(out)->incoming = 1;

   / * Time stamp */
   __net_timestamp(out);

   return 0;
} « end l2cap_over_uwb_extract_connection_info »

/ **************************************************
* l2cap_over_uwb_add_connection_info()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct hci_conn *conn - structure containing connection info
* struct sk_buff *in - structure containing the data packet
* struct sk_buff *out - data packet with connection info added
*
* return - 0 on success, else negative
**************************************************/
int l2cap_over_uwb_add_connection_info(struct hci_conn *conn, struct sk_buff *in, struct sk_buff *out)
{
   struct l2cap_conn *l2conn = (struct l2cap_conn *) conn->l2cap_data;
```

```
    struct l2cap_over_uwb_header *puthdr;

    PDEBUG("Make space in new buffer\n");
    skb_put(out, in->len + L2CAP_OVER_UWB_HEADER_SIZE);

    PDEBUG("Filling in header\n");
    puthdr = (struct l2cap_over_uwb_header *) out->data;
    puthdr->id = L2CAP_OVER_UWB_ID;
    puthdr->len = __cpu_to_le16((unsigned short)in->len);
    bacpy(&(puthdr->src), l2conn->src);
    bacpy(&(puthdr->dst), l2conn->dst);
    puthdr->type = bt_cb(in)->pkt_type;

    PDEBUG("Transfer the data\n");
    memcpy(&(out->data[L2CAP_OVER_UWB_HEADER_SIZE]), in->data, in->len);

    PDEBUG("Done with adding header\n");

    return 0;
} « end l2cap_over_uwb_add_connection_info »

/ *************************************************
* l2cap_over_uwb_bottom_exit()
*
* Function for entering a data packet received from upper layer
* Bluetooth stack into the lower layer UWB routing
*
* struct hci_conn *conn - structure containing connection info
* struct sk_buff *TxSkb - structure containing the data packet
*
* return - 0 on success, else negative
*************************************************/
int l2cap_over_uwb_bottom_exit(struct hci_conn *conn, struct sk_buff *TxSkb)
{
    struct sk_buff *new_skb = 0;
    int ret;

    PDEBUG("Sending data down to UWB_Router [%d]: ", TxSkb->len);
    l2cap_over_uwb_print_skb(TxSkb);

    if(l2cap_over_uwb_external_transmitter == NULL)
    {
            BDEBUG("Error! No receiver function registered to service transmit packet! \n");
            return - 1;
    }
    else
    {
            PDEBUG("Add HCI connection info as header to the packet\n");

            PDEBUG("Allocating buffer\n");
            new_skb = alloc_skb(TxSkb->len + L2CAP_OVER_UWB_HEADER_SIZE + UWB_ROUTER_HEADER_SIZE, GFP_ATOMIC);
            if (! new_skb)
            {
                    BDEBUG("Could not allocate out skb! \n");
                    return - 5;
            }

            PDEBUG("Reserve space for UWB Router header\n");
            skb_reserve(new_skb, UWB_ROUTER_HEADER_SIZE);

            ret = l2cap_over_uwb_add_connection_info(conn, TxSkb, new_skb);
            if(ret)
            {
                    BDEBUG("Error adding connection info");
                    return ret;
            }

            PDEBUG("Calling external trasnmitter function with packet[%d]: ", new_skb->len);
            l2cap_over_uwb_print_skb(new_skb);

            l2cap_over_uwb_external_transmitter(new_skb, INS_MODE_L2CAP);
            kfree_skb(new_skb);
            kfree_skb(TxSkb);
            return 0;
    } « end else »
} « end l2cap_over_uwb_bottom_exit »

/ * Declare this function here to keep Mr. Compiler happy */
static int l2cap_recv_acldata(struct hci_conn *hcon, struct sk_buff *skb, u16 flags);

/ *************************************************
* l2cap_over_uwb_bottom_entry()
*
* Function for entering a data packet received from lower layer
* UWB routing into the upper layer Bluetooth stack
*
* struct sk_buff *RxSkb - structure containing the data
*
* return - 0 on success, else negative
*************************************************/
int l2cap_over_uwb_bottom_entry(struct sk_buff *RxSkb)
{
    struct sk_buff *new_skb = 0;
    struct hci_conn *conn = 0;
    int ret;

    PDEBUG("Allocating buffer\n");
    new_skb = alloc_skb(RxSkb->len - L2CAP_OVER_UWB_HEADER_SIZE, GFP_ATOMIC);
    if (! new_skb)
    {
            BDEBUG("Could not allocate out skb! \n");
```

```
        return - 1;
    }

    ret = l2cap_over_uwb_extract_connection_info(&conn, RxSkb, new_skb);
    if(ret)
    {
        BDEBUG("Error adding connection info");
        return ret;
    }

    PDEBUG("Sending data up to Bluetooth stack [%d]: ", new_skb->len);
    l2cap_over_uwb_print_skb(new_skb);

    PDEBUG("Calling internal receiver function! \n");
    l2cap_recv_acldata(conn, new_skb, 0x2);
    / / kfree_skb(new_skb);
    return 0;
} « end l2cap_over_uwb_bottom_entry »

EXPORT_SYMBOL(l2cap_over_uwb_bottom_entry);

/ ***************************************************************************************************/
/ ***************************************************************************************************/
#endif

static inline int l2cap_do_send(struct sock *sk, struct msghdr *msg, int len)
{
    struct l2cap_conn *conn = l2cap_pi(sk)->conn;
    struct sk_buff *skb, **frag;
    int err, hlen, count, sent=0;
    struct l2cap_hdr *lh;

    BT_DBG("sk %p len %d", sk, len);

    / * First fragment (with L2CAP header) */
    if (sk->sk_type == SOCK_DGRAM)
        hlen = L2CAP_HDR_SIZE + 2;
    else
        hlen = L2CAP_HDR_SIZE;

    count = min_t(unsigned int, (conn->mtu - hlen), len);
    skb = bt_skb_send_alloc(sk, hlen + count, msg->msg_flags & MSG_DONTWAIT, &err);
    if (! skb)
        return err;

    / * Create L2CAP header */
    lh = (struct l2cap_hdr *) skb_put(skb, L2CAP_HDR_SIZE);
    lh->cid = cpu_to_le16(l2cap_pi(sk)->dcid);
    lh->len = cpu_to_le16(len + (hlen - L2CAP_HDR_SIZE));

    if (sk->sk_type == SOCK_DGRAM)
        put_unaligned(l2cap_pi(sk)->psm, (__le16 *) skb_put(skb, 2));
    if (memcpy_fromiovec(skb_put(skb, count), msg->msg_iov, count)) {
        err = - EFAULT;
        goto fail;
    }

    sent += count;
    len -= count;

    / * Continuation fragments (no L2CAP header) */
    frag = &skb_shinfo(skb)->frag_list;
    while (len) {
        count = min_t(unsigned int, conn->mtu, len);
        *frag = bt_skb_send_alloc(sk, count, msg->msg_flags & MSG_DONTWAIT, &err);
        if (! *frag)
            goto fail;

        if (memcpy_fromiovec(skb_put(*frag, count), msg->msg_iov, count)) {
            err = - EFAULT;
            goto fail;
        }

        sent += count;
        len -= count;
        frag = &(*frag)->next;
    }

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
    / ***************************************************************************************************
    * BToUWB Insertion
    ***************************************************************************************************/
    if((l2cap_over_uwb_external_transmitter ! = NULL) && (l2cap_over_uwb_routing_mode))
    {
        PDEBUG("\n <- - - - Sending ACL data over UWB router - - - - >\n");
        bt_cb(skb)->pkt_type = HCI_ACLDATA_PKT;
        if ((err = l2cap_over_uwb_bottom_exit(conn->hcon, skb)) < 0)
            goto fail;
    }
    else
    {
        PDEBUG("\n <- - - - Sending ACL data over Bluetooth link - - - - >\n");
        if ((err = hci_send_acl(conn->hcon, skb, 0)) < 0)
            goto fail;
    }
    / ***************************************************************************************************/
#else
        if ((err = hci_send_acl(conn->hcon, skb, 0)) < 0)
            goto fail;
#endif
```

```
    return sent;

    fail:
            kfree_skb(skb);
    return err;
} « end l2cap_do_send »
```

D.4.2BlueZ sco.c modifications

```
#define UWB_ROUTER_CONNECT_TO_BLUEZ 1
#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
/ ****************************************************************************************************
* BToUWB Insertion
***************************************************************************************************/
#include "../ ../ uwb_router/ uwb_router_src.h"

#define SCO_OVER_UWB_STACK_DEBUG 1
/ /#define SCO_OVER_UWB_PACKET_DEBUG 1

#undef BDEBUG / * undef it, just in case */
#ifdef SCO_OVER_UWB_STACK_DEBUG
#define BDEBUG(fmt, args...) printk(KERN_ERR "SCO> "); printk(KERN_ERR fmt, ## args)
#else
#define BDEBUG(fmt, args...)
#endif

#undef PDEBUG / * undef it, just in case */
#ifdef SCO_OVER_UWB_PACKET_DEBUG
#define PDEBUG(fmt, args...) printk(KERN_ERR "SCO> "); printk(KERN_ERR fmt, ## args)
#else
#define PDEBUG(fmt, args...)
#endif

/ *************************************************
* sco_over_uwb_print_skb()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct sk_buff *skb_ptr - structure containing the data
*
* return - number of bytes printed
*************************************************/
ssize_t sco_over_uwb_print_skb(struct sk_buff *skb_ptr)
{
    int i = 0;
#ifdef SCO_OVER_UWB_PACKET_DEBUG
    for(i=0; i<skb_ptr->len; i++)
    {
            printk(KERN_ERR "%02X ", (unsigned char)skb_ptr->data[i]);
    }
    printk(KERN_ERR "\n");
#endif
    return i;
}

static char sco_over_uwb_routing_mode = 0;

/ *************************************************
* sco_over_uwb_set_routing_mode()
*
* Function for enabling / disabling the routing of packets over
* the UWB Routing subsystem
*
* char mode - signifies the requested mode
*
* return - void
*************************************************/
void sco_over_uwb_set_routing_mode(char mode)
{
    BDEBUG("SCO setting routing mode to %d\n", mode);
    sco_over_uwb_routing_mode = mode;
}

EXPORT_SYMBOL(sco_over_uwb_set_routing_mode);

struct sk_buff_head sco_over_uwb_rx_q;
static atomic_t sco_over_queue_length;
/ * Pointer to lower layer function that handles data packets to be transmitted */
static int (*sco_over_uwb_external_transmitter) (struct sk_buff *data, char mode) = NULL;

/ *************************************************
* sco_over_uwb_register_transmitter_function()
*
* Registers a pointer to a lower layer function to be called
* that should handle packets transmitted from the upper
* level Bluetooth stack.
*
* void* transmitter_function - function pointer to register
*
* return - always 0
*************************************************/
int sco_over_uwb_register_transmitter_function(void* transmitter_function)
{
    BDEBUG("Registering transmitter function\n");
    sco_over_uwb_external_transmitter = transmitter_function;
    if(transmitter_function == NULL)
    {
            / * Purge wait queue */
            BDEBUG("Purge wait queue of length %d\n", sco_over_uwb_rx_q.qlen);
            skb_queue_purge(&sco_over_uwb_rx_q);
    }
    else
    {
            / * Setup wait queue */
            BDEBUG("Setup wait queue\n");
            skb_queue_head_init(&sco_over_uwb_rx_q);
            sco_over_queue_length.counter = 0;
    }
    return 0;
} « end sco_over_uwb_register_transmitter_function »

/ * Export symbol so that external modules can find it */
EXPORT_SYMBOL(sco_over_uwb_register_transmitter_function);

#define SCO_OVER_UWB_ID 'h'
#define SCO_OVER_UWB_HEADER_SIZE 3

struct sco_over_uwb_header
{
    __u8 id;
    __u16 len;
}__attribute__ ((packed));

/ **************************************************
* sco_over_uwb_extract_connection_info()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct hci_conn *conn - structure containing connection info
* struct sk_buff *in - data packet with connection info added
* struct sk_buff *out - data packet with connection info removed
*
* return - 0 on success, else negative
**************************************************/
int sco_over_uwb_extract_connection_info(struct sk_buff *in, struct sk_buff *out)
{
    struct sco_over_uwb_header *hdr = (struct sco_over_uwb_header *) in->data;
    int new_length;

    / * Check ID */
    if(hdr->id != SCO_OVER_UWB_ID)
    {
            BDEBUG("Unexpected ID in header! \n");
            return - 1;
    }

    new_length = (int)__le16_to_cpu(hdr->len);
    if(new_length ! = (in->len - SCO_OVER_UWB_HEADER_SIZE))
    {
            BDEBUG("Unexpected length %d in header, needed %d! \n", new_length, (in->len - SCO_OVER_UWB_HEADER_SIZE));
            return - 2;
    }

    if (in->len) {
            memcpy(skb_put(out, new_length), &(in->data[SCO_OVER_UWB_HEADER_SIZE]), new_length);
    }

    return 0;
} « end sco_over_uwb_extract_connection_info »

/ **************************************************
* sco_over_uwb_add_connection_info()
*
* Function for printing the contents of a buffer for debug purpouses
*
* struct hci_conn *conn - structure containing connection info
* struct sk_buff *in - structure containing the data packet
* struct sk_buff *out - data packet with connection info added
*
* return - 0 on success, else negative
**************************************************/
int sco_over_uwb_add_connection_info(struct sk_buff *in, struct sk_buff *out)
{
    struct sco_over_uwb_header *puthdr;

    PDEBUG("Make space in new buffer\n");
    skb_put(out, in->len + SCO_OVER_UWB_HEADER_SIZE);

    PDEBUG("Filling in header\n");
    puthdr = (struct sco_over_uwb_header *) out->data;
    puthdr->id = SCO_OVER_UWB_ID;
    puthdr->len = __cpu_to_le16((unsigned short)in->len);

    PDEBUG("Transfer the data\n");
    memcpy(&(out->data[SCO_OVER_UWB_HEADER_SIZE]), in->data, in->len);

    PDEBUG("Done with adding header\n");

    return 0;
}

/ **************************************************
* sco_over_uwb_frame_replace()
*
* Function for replacing the data inside a synchronous SCO
* data packet received with the data obtained from an
* asynchronous packet received over the UWB interface
*
* struct sk_buff *dst - target buffer to be replaced
*
* return - 0 on success, else negative
***************************************************/
int sco_over_uwb_frame_replace(struct sk_buff *dst)
{
    int i;
    int source_ready = 0;
    int start_offset = 0;
    struct sk_buff *src = 0;
    static int counter = 0;

    PDEBUG("BT: Received packet %d with length %d and wait queue %d\n", counter, dst->len, atomic_read(&(sco_over_queue_length)));
    counter++;

    if(atomic_read(&(sco_over_queue_length)))
    {
        src = skb_dequeue(&sco_over_uwb_rx_q);
        atomic_dec(&(sco_over_queue_length));
        source_ready = 1;
    }
    else
    {
        PDEBUG("Queue dont exist, length is %d! \n", atomic_read(&(sco_over_queue_length)));
        source_ready = 0;
    }

    / * Check details */
    if((source_ready) && (! src)) {
        PDEBUG("Source buffer dont exist! \n");
        source_ready = 0;
    }

    if(! dst) {
        PDEBUG("Destination buffer dont exist! \n");
        return - 1;
    }

    if((source_ready) && (! src->len)) {
        BDEBUG("Source length dont exist! \n");
        source_ready = 0;
    }

    if(source_ready)
    {
        PDEBUG("Popped frame with length %d from wait queue\n", src->len);
    }

    if(! dst->len) {
        BDEBUG("Destination length dont exist! \n");
        return - 2;
    }

    if((source_ready) && (src->len < start_offset)) {
        BDEBUG("Source length %d shorter than minimum %d! \n", src->len, start_offset);
        source_ready = 0;
    }

    if(dst->len < start_offset) {
        BDEBUG("Destination length %d shorter than minimum %d! \n", dst->len, start_offset);
        return - 3;
    }

    if((source_ready) && (src->len < dst->len)) {
        BDEBUG("Source length %d shorter than destination length %d! \n", src->len, dst->len);
        source_ready = 0;
    }

    if(source_ready == 1)
    {
        PDEBUG("Raplacing %d bytes @ run %d\n", dst->len, count++);
        for(i=start_offset;i<dst->len;i++)
        {
            dst->data[i] = src->data[i];
        }
        kfree_skb(src);
    }
    else
    {
        PDEBUG("Raplacing %d bytes @ run %d with zeros\n", dst->len, count++);
        for(i=start_offset;i<dst->len;i++)
        {
            dst->data[i] = 0;
        }
    }

    return source_ready - 1;
} « end sco_over_uwb_frame_replace »

/ ***************************************************
* sco_over_uwb_bottom_exit()
*
* Function for entering a data packet received from upper layer
* Bluetooth stack into the lower layer UWB routing
*
* struct hci_conn *conn - structure containing connection info
* struct sk_buff *TxSkb - structure containing the data packet
*
```

```
* return - 0 on success, else negative
**************************************************/
int sco_over_uwb_bottom_exit(struct hci_conn *conn, struct sk_buff *TxSkb)
{
    struct sk_buff *new_skb = 0;
    int ret;

    PDEBUG("Sending data down to UWB_Router [%d]: ", TxSkb->len);
    sco_over_uwb_print_skb(TxSkb);

    if(sco_over_uwb_external_transmitter == NULL)
    {
            BDEBUG("Error! No receiver function registered to service transmit packet! \n");
            return - 1;
    }
    else
    {
            PDEBUG("Allocating buffer\n");
            new_skb = alloc_skb(TxSkb->len + SCO_OVER_UWB_HEADER_SIZE + UWB_ROUTER_HEADER_SIZE, GFP_ATOMIC);
            if (! new_skb)
            {
                    BDEBUG("Could not allocate out skb! \n");
                    return - 5;
            }

            PDEBUG("Reserve space for UWB Router header\n");
            skb_reserve(new_skb, UWB_ROUTER_HEADER_SIZE);

            ret = sco_over_uwb_add_connection_info(TxSkb, new_skb);
            if(ret)
            {
                BDEBUG("Error adding header");
                return ret;
            }

            PDEBUG("Calling external trasnmitter function with packet[%d]: ", new_skb->len);
            sco_over_uwb_print_skb(new_skb);
            sco_over_uwb_external_transmitter(new_skb, INS_MODE_SCO);
            kfree_skb(new_skb);
            return 0;
    } « end else »
} « end sco_over_uwb_bottom_exit »

/ **************************************************
* sco_over_uwb_bottom_entry()
*
* Function for entering a data packet received from lower layer
* UWB routing into the upper layer Bluetooth stack
*
* struct sk_buff *RxSkb - structure containing the data
*
* return - 0 on success, else negative
**************************************************/
int sco_over_uwb_bottom_entry(struct sk_buff *RxSkb)
{
    struct sk_buff *new_skb = 0;
    int ret;

    PDEBUG("Allocating buffer\n");
    new_skb = alloc_skb(RxSkb->len - SCO_OVER_UWB_HEADER_SIZE, GFP_ATOMIC);
    if (! new_skb)
    {
            BDEBUG("Could not allocate out skb! \n");
            return - 1;
    }

    ret = sco_over_uwb_extract_connection_info(RxSkb, new_skb);
    if(ret)
    {
            BDEBUG("Error removing connection info");
            return ret;
    }

    / * Queue the buffer for processing */
    skb_queue_tail(&sco_over_uwb_rx_q, new_skb);
    atomic_inc(&(sco_over_queue_length));

    return 0;
} « end sco_over_uwb_bottom_entry »

EXPORT_SYMBOL(sco_over_uwb_bottom_entry);

/ *****************************************************************************************************/
/ *****************************************************************************************************/
#endif


static inline int sco_send_frame(struct sock *sk, struct msghdr *msg, int len)
{
    struct sco_conn *conn = sco_pi(sk)->conn;
    struct sk_buff *skb;
    int err, count;

    / * Check outgoing MTU */
    if (len > conn->mtu)
            return - EINVAL;
    BT_DBG("sk %p len %d", sk, len);
    count = min_t(unsigned int, conn->mtu, len);
    if (! (skb = bt_skb_send_alloc(sk, count, msg->msg_flags & MSG_DONTWAIT, &err)))
            return err;
    if (memcpy_fromiovec(skb_put(skb, count), msg->msg_iov, count)) {
        err = - EFAULT;
        goto fail;
    }

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
    / ****************************************************************************************************
    * BToUWB Insertion
    ****************************************************************************************************/

    if((sco_over_uwb_external_transmitter ! = NULL) && (sco_over_uwb_routing_mode))
    {
        PDEBUG("\n <- - - - Sending SCO data over UWB router - - - - >\n");
        sco_over_uwb_bottom_exit(conn->hcon, skb);
    }

    PDEBUG("\n <- - - - Sending SCO data over Bluetooth link - - - - >\n");
#endif

    if ((err = hci_send_sco(conn->hcon, skb)) < 0)
        return err;

    return count;
    fail:
        kfree_skb(skb);
    return err;
} « end sco_send_frame »

static int sco_recv_scodata(struct hci_conn *hcon, struct sk_buff *skb)
{
    struct sco_conn *conn = hcon->sco_data;

    PDEBUG("<- - sco_recv_scodata - ->\n");
    if (! conn)
    {
        BDEBUG("! Conn\n");
        goto drop;
    }

    BT_DBG("conn %p len %d", conn, skb->len);

#ifdef UWB_ROUTER_CONNECT_TO_BLUEZ
    / ****************************************************************************************************
    * BToUWB Insertion
    ****************************************************************************************************/
    if((sco_over_uwb_external_transmitter ! = NULL) && (sco_over_uwb_routing_mode))
    {
        / * Replace the frame contents */
        sco_over_uwb_frame_replace(skb);
    }
#endif

    if (skb->len) {
        sco_recv_frame(conn, skb);
        return 0;
    }

    drop:
        kfree_skb(skb);
    return 0;
} « end sco_recv_scodata »
```