



Java Bluetooth stack Technical Documentation

Version 1.0

Doc. No.:

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Revision History

Date	Version	Description	Author
2004-01-10	0.1	Initial Draft	Marko Đurić
2004-01-11	0.4	Added Installation guides and chapter about using CVS from Eclipse Platform	Tomislav Sečen
2004-01-11	0.5	Minor updates	Marko Đurić
2004-01-14	0.8	Added L2CAP, SDP, RFCOMM and OBEX to the chapter "Bluetooth technology overview" and added "JSR-82 overview" chapter	Marko Đurić
2004-01-15	1.0	Minor updates	Marko Đurić

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Table of Contents

1.	Introduction	8
1.1	Purpose of this document	8
1.2	Intended Audience	8
1.3	Scope	8
1.4	Definitions and acronyms	8
1.4.1	Definitions	8
1.4.2	Acronyms and abbreviations	8
2.	Acknowledgements	10
3.	Motivation	10
4.	Bluetooth technology overview	10
4.1	Harald stack	10
4.2	General Description	11
4.3	Network topology	11
4.4	Bluetooth SIG	12
4.5	Where can Bluetooth be useful?	12
4.6	Layers	12
4.6.1	Baseband	13
4.6.2	Link Control (LC)	14
4.6.3	Link Manager (LM)	14
4.6.4	Host Controller Interface (HCI)	15
4.6.5	Transport Interface	15
4.6.6	Logical Link Control and Adaptation Protocol (L2CAP)	17
4.6.7	Service Discovery Protocol (SDP)	19
4.6.8	Serial-port emulation protocol (RFCOMM)	20
4.6.9	OBject EXchange protocol (OBEX)	22
4.7	Bluetooth security	25
4.7.1	Authentication	25
4.7.2	Encryption	26
5.	Java Bluetooth Stack – package overview	28
5.1	Host Controller Interface (HCI)	Error! Bookmark not defined.
5.2	Logical Link Control and Adaptation Protocol (L2CAP)	Error! Bookmark not defined.
5.3	Service Discovery Protocol (SDP)	Error! Bookmark not defined.
5.4	Stack	Error! Bookmark not defined.
5.5	microedition.io package	Error! Bookmark not defined.
5.6	javax.bluetooth package	Error! Bookmark not defined.
5.7	hr.fer.rasip.bluesec package (BCC – Bluetooth Control Center)	28
5.8	hr.fer.rasip.rfcomm package	34
5.9	javax.obex package	39
5.10	hr.fer.rasip.obex package	41
5.11	Distributed	Error! Bookmark not defined.
5.12	Debug	Error! Bookmark not defined.
6.	JSR-82 overview	42
6.1	Device management	43
6.2	Discovery	44
6.3	Communication	45
7.	Installation guide – software	47

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

7.1 Installing Eclipse Platform and Java commAPI and configuring environment variables for commAPI	48
7.1.1 Eclipse Platform installation	48
7.1.2 Sun Wireless ToolKit installation	48
7.1.3 Installing commAPI	48
7.1.4 Importing project into workspace	48
7.1.5 Running samples and JUnit tests	48
8. Installation guide – hardware	48
8.1 Xircom CBT	49
8.2 Generic Bluetooth serial module install instructions	50
9. Conclusion	Error! Bookmark not defined.
Appendix A: Getting commAPI to work	52
Appendix B: Importing existing projects into workspace	54
Appendix C: Creating Eclipse Runtime configurations	55
Appendix D: Checking and configuring project references	56
Appendix E: Eclipse JUnit support	57
Writing JUnit tests:	57
Running JUnit tests	60
Creating a new JUnit Run Configuration	61
Creating a JUnit test suite	62
Appendix F: Eclipse & CVS	64
Creating a CVS repository location	64
Sharing a new project using CVS	65
Checking out a project from a CVS repository	67
Updating	68
Committing	72
Synchronizing with a CVS repository	73
Literature and resources	73
Literature and resources	74
Books	74
Links on the Internet	74
Index	74

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Table of Figures

Fig. 4.3.2: More Piconets forms a Scatternet.....	12
Fig. 4.6.1: Layered structure of Bluetooth stack	13
Fig. 4.6.2: Standard packet format	13
Fig. 4.6.3: Connection establishment.....	14
Fig. 4.6.4: Bluetooth Hardware architecture overview	15
Fig. 4.6.5: Data flow from one Bluetooth device to another, using USB transport interface	16
Fig. 4.6.6: HCI UART transport interface.....	16
Fig. 4.6.7: L2CAP in the Bluetooth protocol architecture	17
Fig. 4.6.8: L2CAP Packet - Connection-oriented channel.....	18
Fig. 4.6.9: L2CAP Packet – Connection-less channel	18
Fig. 4.6.10: Service Discovery Protocol.....	19
Fig. 4.6.11: Protocol Data Unit (PDU format)	19
Fig. 4.6.12: Multiple emulated serial ports	20
Fig. 4.6.13: Emulating serial ports coming from two Bluetooth devices.....	20
Fig. 4.6.14: RFCOMM reference model.....	21
Fig. 4.6.15: The RFCOMM communication model	22
Fig. 4.6.16: OBEX hierarchy diagram	23
Fig. 4.7.1: Authentication procedure.....	25
Fig. 4.7.2: Authentication.....	26
Fig. 4.7.3: Stream ciphering for Bluetooth with E0.....	26
Fig. 4.7.4: Functional description of encryption procedure.....	27
Fig. 4.7.5: Generating encryption key using E3.....	27
Fig. 5.7.1: class diagram for implementation of Bluetooth security methods.....	28
Fig. 5.7.2: Sequence diagram for authentication procedure	30
Fig. 5.7.3: BCC - encrypted connection sequence diagram.....	33
Fig. 5.7.4: BCC - Client security	34
Fig. 5.8.1: RFCOMM class diagram.....	35
Fig. 5.8.2: The format of each RFCOMM frame.....	36
Fig. 5.8.3: RFCOMM address field.....	36
Fig. 5.8.4: Control field.....	36
Fig. 5.8.5: Length field	37
Fig. 5.8.6: RFCOMM sequence diagram.....	38
Fig. 5.9.1: javax.obex package class diagram	39
Fig. 5.10.1: hr.fer.rasip.obex package class diagram	41
Fig. 5.10.2: hr.fer.rasip.obex package sequence diagram	42
Fig. 6.0.1: JSR-82 API package structure.....	43
Fig. 8.5: ROK 104001 Module installation	51
Fig. A.1: Blackbox example main screen	53
Fig. C.1: Creating the new Runtime Configuration	55
Fig. D.1: Project's properties – Java Build Path.....	56
Fig. E.1: Opening project's properties.....	57
Fig. E.2: Adding <i>junit.jar</i> to the build class-path	58
Fig. E.3: Using wizard for writing new JUnit TestCase or TestSuite.....	59
Fig. E.4: Creating a new JUnit TestCase.....	59
Fig. E.5: JUnit test run status: a) test failed, b) test successfully completed	60
Fig. E.6: Creating new JUnit run configuration.....	61
Fig. E.7: JUnit Run test created and ready to run.....	62
Fig. E.8: Creating a new JUnit TestSuite	63
Fig. F.1: Adding the CVS repository directory.....	65
Fig. F.2: Sharing a project using CVS.....	66
Fig. F.3: Sharing an existing CVS repository	67
Fig. F.4: Creating a local project from CVS repository.....	68
Fig. F.5: Updating the local project.....	69

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Fig. F.6: Committing the local changes..... 70

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

1. Introduction

1.1 Purpose of this document

Technical Documentation provides technical information about Java Bluetooth Stack. It gives some basic introduction to the Bluetooth technology, and the technical guidance for using the Java Bluetooth Stack.

1.2 Intended Audience

This document is intended for Java Bluetooth developers or anyone interested in using Java Bluetooth stack.

1.3 Scope

Bluetooth technology for wireless communication is described by the layers of the Bluetooth stack and by the basic principles of the Bluetooth communications.

Java Bluetooth Stack is introduced to the users/developers package-by-package, so it can be easily used later in developing Bluetooth java applications.

1.4 Definitions and acronyms

1.4.1 Definitions

Keyword	Definitions
Piconet	More Bluetooth devices organized in a group
Scatternet	Formation of more piconets connected together in one network
JSR-82	Specification that defines Java Bluetooth API

1.4.2 Acronyms and abbreviations

Acronym or abbreviation	Definitions
LC	Link Controller
LM	Link Manager
LMP	Link Manager Protocol
HCI	Host Controller Interface
ACL	Asynchronous Connection-Less
SCO	Synchronous Connection-Oriented
L2CAP	Logical Link Control and Adaptation Protocol
SDP	Service Discovery Protocol
RFCOMM	Radio-Frequency COMMunication
OBEX	OBject EXchange protocol
BCC	Bluetooth Control Center
PSM	Protocol/Service Multiplexer
DLC	Data Link Connection
PDU	Protocol Data Unit
API	Application Programming Interface
JSR	Java Specification Request
CLDC	Connected, Limited Device Configuration
MIDP	Mobile Information Device Profile
GAP	General Access Profile

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

SDAP	Service Discovery Application Profile
J2ME	Java 2 Micro Edition
J2SE	Java 2 Standard Edition
SPP	Serial Port Profile
GCF	Generic Connection Framework
UUID	Universal Unique Identifier
MTU	Maximum Transmission Unit
GOEP	Generic Object Exchange Profile

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

2. Acknowledgements

This project wouldn't have the form it has today if it weren't for two great Bluetooth stacks: [Harald's Bluetooth stack](#), created by John Eker, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, and [javablueetooth](#), created by Christian Lorentz. This project is based on javablueetooth, and it uses some code from Harald. We would like to thank both of them for creating these great stacks, and for teaching us a dozen of neat in-depth Java tricks.

Another great help in coding of RFCOMM protocol layer was the official Linux Bluetooth protocol stack, [BlueZ](#), so we would like to thank them also.

3. Motivation

At the time this project was started (2003/11), there was no open-source (nor even freeware) Java Bluetooth stacks available. Java developer interested in using Bluetooth wireless technology in his applicatinos had two options:

- buy commercial stack (library) – you become bound to a single manufacturer, and at his mercy for updates
- use BlueZ, a free-source official Linux Bluetooth stack written in C, with it's JNI wrapped Java brother, JBlueZ

To change this, this project was started, and it's goal is to create a full JSR-82 compliant Bluetooth stack.

4. Bluetooth technology overview

4.1 Harald stack

A little bit of history?

Harald Bluetooth (originally Blåtand) was a Viking king who ruled all of Denmark and Norway in the 10th century (960 – 986).

Harald Blåtand was a very active king. He won the whole of Denmark and Norway and large enterprises were commenced during his time as king; much of Harald's history was learnt from two runic stones erected in the town of Jelling in Denmark.

Did he actually have blue teeth, you must wonder? No, Blåtand actually means dark complexion – he had very dark hair, which was unusual for Vikings. Not only did Harald not fit the classic image physically, he was a rather unusual Viking.

That is, if your understanding was that the life of a Viking was all battles and pillage. The good King Harald brought Christianity to Scandinavia and also united Denmark and Norway.

When he was first crowned in 960, he erected a runic stone – monument to his parents – with the following text written (in symbols known as runes) on it: "Harald king executes these sepulchral monuments after Gorm, his father and Thyra, his mother. The Harald who won the



Fig. 4.1.1: Runic stone in memory of Harald's parents

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

whole of Denmark and Norway and turned the Danes to Christianity."

A millennium later, in 1994 in Lund, Sweden, a new wireless technology for mobile communication was named after the great king, and his name became famous worldwide.

4.2 General Description

Bluetooth is wireless communication protocol that was originally invented 1994. by Swedish phone equipment maker Ericsson. Its purpose is wireless communication between mobile devices on short distances (up to 100m).

Bluetooth operates in 2.4GHz radio band that is reserved for Industrial – Scientific – Medical (ISM) purposes. Because it communicates by radio signals, there is no need for optical visibility between devices.

There are 79 available RF channels with base frequency $f = (2402+k)\text{MHz}$, $k=0,1,\dots,78$. Channel spacing is 1MHz, and in order to comply Bluetooth frequency range of 2400MHz – 2483.5MHz, there is 2MHz Lower Guard Band, and 3,5MHz Upper Guard Band. For added security, channels use frequency hopping (frequency is changed in pseudo-random pattern 1600 times per second).

Some countries have national limitations in the frequency range, so the Bluetooth frequency range is reduced in those countries. Because of that, products implementing the reduced frequency band **will not work** with products implementing the full band.

There may be some questions why use Bluetooth, when there is existing Infrared or 802.11b wireless communication standards.

When comparing with Infrared communication, the answer is pretty simple – Infrared must have optical visibility between the devices, and that's the main disadvantage. For that reason, it's better to deal with comparison of Bluetooth and 802.11b wireless protocols.

The main difference between Bluetooth and 802.11b is the different goals of those two communication protocols. While 802.11b is intended to be used for connecting relatively large devices that uses lots of power at high speeds (11Mbps), Bluetooth is the complete opposite of that – it is used for small (mobile) devices that runs on batteries (or other low-power supply), and because of that, for lower speed communications on short distances (10m (=30ft) at max of 1Mbps).

4.3 Network topology

Intention of Bluetooth devices is ad-hoc connectivity of small devices like mobile phones or PDA-s and for that reason the range is limited to 10m (max. of 100m in Class I devices – larger devices with better power supply (more capacity)). It is mostly used for small amount of data transfer between devices (asynchronous mode) or for speech (synchronous mode).

Bluetooth connection philosophy is very simple – mobile devices that wants to communicate via Bluetooth connection must be able to create small networks with minimum of user interaction (principle of ad-hoc networking).

Ad-hoc networking by definition means that communicating devices can spontaneously form a community of networks that persists only as long as it's needed. Other RF networking (802.11b for example) needs user interaction for creating and administrating the network.

Two or more Bluetooth-enabled devices sharing the same channel are organized in groups. This groups are called *piconets* and they can contain maximum of eight devices (one master and up to seven active slaves). A master unit is the device that initiates the communication.

A device in one piconet can communicate to other device in another piconet, and with that connection it forms a *scatternet*. Device that is slave in one piconet can be master in another piconet.

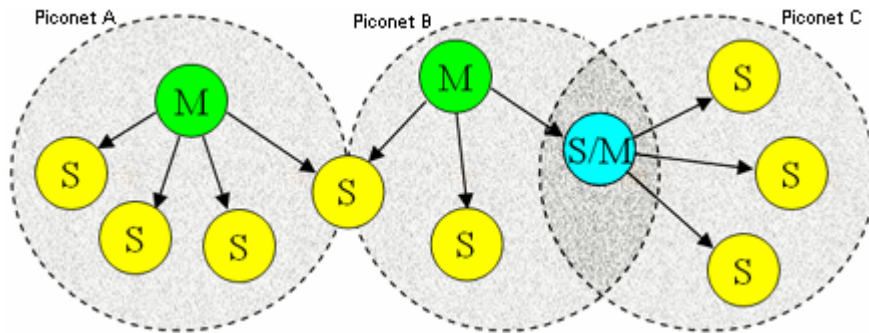


Fig. 4.3.2: More Piconets forms a Scatternet

The normal duration of transmission is one time slot (625 μ s), and a packet can last up to five time slots in length. For achieving full-duplex communication, Bluetooth uses TDM (time-division multiplexing) scheme, in which a master device always uses an even-numbered slot, and slave uses an odd-numbered slot.

4.4 Bluetooth SIG

In 1997, some major companies (such as IBM, Intel, Nokia and Toshiba) joined with Ericsson into Bluetooth Special Interest Group (SIG) consortium for further development of Bluetooth protocol. After the release of the first Bluetooth specification (version 1.0), 3COM, Agere, Microsoft and Motorola also joined the SIG consortium. Bluetooth SIG now has more than 2000 members.

Bluetooth SIG is responsible for updating the Bluetooth specifications and for promoting and improving the Bluetooth standard.

4.5 Where can Bluetooth be useful?

Bluetooth is useful for connecting low-power mobile devices on short ranges. Here are some of the main advantages of Bluetooth:

- communication with peripherals (i.e. computer -> printer communication, etc.)

- very low power needed for Bluetooth communications (1mW on Class 3 devices)

- it is specially designed for smaller data transfers (synchronization, for example)

- Bluetooth is inexpensive (relatively low cost of incorporating Bluetooth technology to existing PDA-s, cell-phones, etc.)

- it uses regulated, but unlicensed radio frequency band (ISM band)

- communicating devices don't need an unobstructed line of sight between them

- Bluetooth uses frequency hopping so there is very low possibility that communications will be intercepted

- Bluetooth can handle both voice and data communications

Because of these advantages, there is a lot of possibilities for using Bluetooth communication. It can be used anywhere where is need for transfer of small amount of data on shorter ranges (synchronization, for example) and it can be also used as a cable replacement technology.

Bluetooth is not useful for large file transfers or long-range communication, but that's not the intention of Bluetooth at all. For that purposes there is 802.11b protocol (or other versions of that protocol (802.11a, 802.11g)).

4.6 Layers

Bluetooth stack has a layered structure (like referent ISO/OSI network model). It consists of protocols that are specific to Bluetooth (L2CAP, SDP, etc.) and other adopted protocols (i.e. OBEX).

There are four main groups in Bluetooth stack:

- Bluetooth core protocols – Baseband, Link Manager Protocol, L2CAP and SDP
- Cable replacement protocol – RFCOMM
- Telephony control protocol – TCS Binary
- Adopted protocols – PPP, UDP/TCP/IP, OBEX, WAP

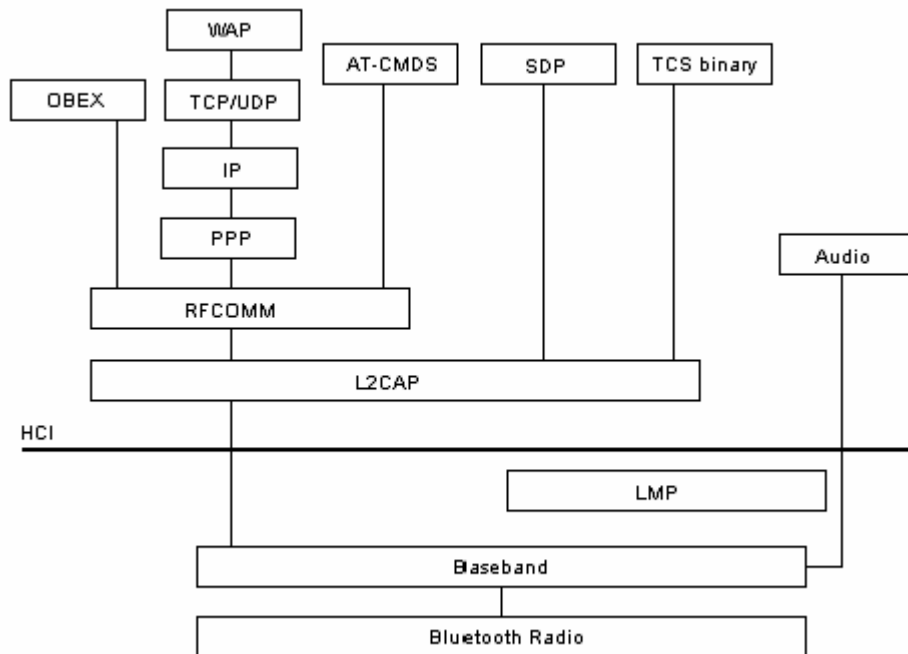


Fig. 4.6.1: Layered structure of Bluetooth stack

4.6.1 Baseband

Bluetooth operates in the unlicensed ISM band at 2.4GHz. It uses binary FM modulation to minimize transceiver complexity, and frequency hop for preventing interference with other devices that are communicating on the same frequency. Communication is full-duplex, and that is achieved by using TDD (time-division duplex) scheme, where time is divided in slots. Every slot has length of 625 μ s (there is 1600 frequency hops in second). Packets are usually sent in single slot, but it can be extended up to 5 slots.

Baseband layer is the Bluetooth layer that is positioned right above the Bluetooth radio. It enables physical radio-frequency (RF) link between Bluetooth enabled devices that wants to communicate. Its function is to manage Bluetooth channels (frequency hopping, time-slots, etc.) and packet transmissions.

Packets defined by Baseband Specification are being sent in Little Endian format. That means the Least Significant Bit (LSB) is first sent over the air. General packet format is shown on Fig. 4.6.2.

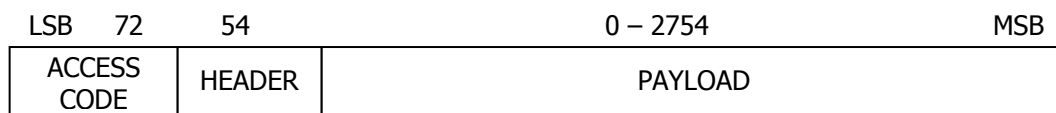


Fig. 4.6.2: Standard packet format

The access code and header size have fixed length (72 bits and 54 bits respectively). The payload can range from 0 to maximum of 2754 bits. Packets may be very short and contain only access code, they can also contain access code and header, and there are full-sized packets that contain access code, header and payload.

For detailed specification of standard packet format, please refer to the Specification of the

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Bluetooth System (Books [1]).

4.6.2 Link Control (LC)

Link Control channel is mapped onto the packet header (see Fig. 4.6.2.). This channel carries low level link control information like ARQ (Automatic Repeat reQuest), flow control, and payload characterization. The LC channel is carried in every packet except in the ID packet that has no packet header.

The Link Control commands allow the Host Controller to control connections to other Bluetooth devices. When the LC commands are used, the Link Manager controls piconet and scatternet creation and management.

4.6.3 Link Manager (LM)

Link Manager Protocol (LMP) messages are used for link set-up, security and control. They are transferred in the payload instead of L2CAP and are distinguished by the L_CH code 11 (that indicates LM channel). LM channel typically uses protected DM (Data Medium rate) packets.

The Link Manager control channel carries control information (LMP messages) exchanged between the link managers of the master and slave(s). Those messages have higher priority than user data, so if the LM needs to send a message, it will not be delayed by the L2CAP traffic. Delay may only occur if there is many retransmissions of individual baseband packets.

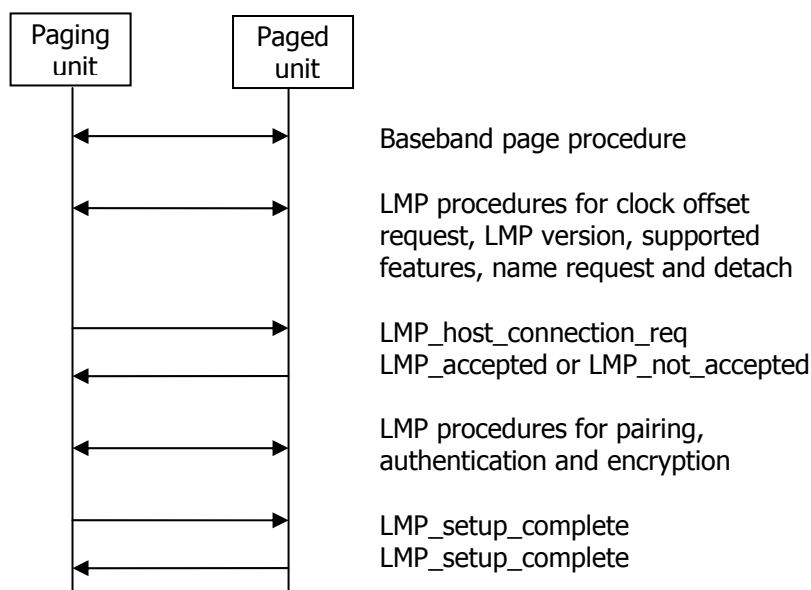


Fig. 4.6.3: Connection establishment

Figure 4.6.3. shows connection establishment between two devices – paging device and paged device. When the paging device wishes to create a connection involving layers above LM, it sends LMP_host_connection_req. Paged device upon receiving connection request can accept or reject that connection. In case it wants to accept the connection, it sends response LMP_accepted to the paging device. After LMP_accepted response, LMP security procedures (pairing, authentication and encryption) can be invoked. After both devices have sent LMP_setup_complete the first packet on a logical channel different from LMP can then be transmitted.

In some cases there is need for slave to request master/slave switch. That request is sent by LMP_slot_offset or LMP_switch_req after the LMP_host_connection_req is received. When the master/slave switch has been successfully completed, the old slave will reply with LMP_accepted, but with the transaction ID set to 0.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

4.6.4 Host Controller Interface (HCI)

HCI provides a uniform interface method of accessing the Bluetooth hardware capabilities. HCI commands are implemented in HCI firmware by accessing baseband commands, link manager commands, hardware status registers, control registers, and event registers. In some cases there is also Host Controller Transport Layer as an intermediate layer for communication between HCI firmware and host HCI driver.

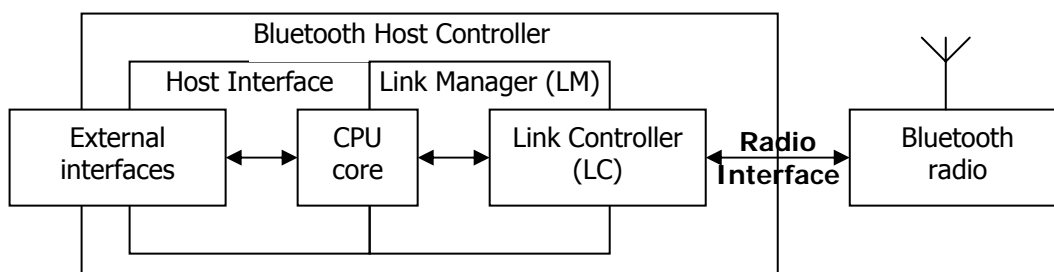


Fig. 4.6.4: Bluetooth Hardware architecture overview

In order to avoid filling up the Host Controller data buffers with ACL (Asynchronous Connection-Less) data destined for a remote device that is not responding, HCI uses flow control in the direction from the Host to the Host Controller.

HCI communicates with Bluetooth hardware using HCI Link commands. These commands provide the Host with the ability to control the link layer connections to the other Bluetooth devices. LM is typically involved with these commands for exchanging LMP messages with other devices.

The results of commands are reported back to the Host in the form of events. This is because HCI commands may take different amounts of time to be completed. For detection of HCI Transport layer errors, the timeout is set between the transmission of the Host's command and the reception of the Host controller's response. Timeout is one second, by default.

4.6.5 Transport Interface

Transport Interface layer is a transport layer between the Host Controller driver and the Host Controller. Some of the examples of transport interfaces are PCMCIA, UART, USB, etc.

This layer is used because the Host Controller driver should not care whether it is running over USB or PC-Card. This allows the Host Controller Interface or the Host Controller to be upgraded without affecting the transport layer.

4.6.5.1 Standard transport interface (UART, USB)

USB transport interface

USB hardware can be embodied as a USB dongle, or it can be integrated onto the motherboard of a notebook PC.

The firmware configuration consists of two interfaces:

0. interface with no alternate settings; it contains bulk and interrupt end-point
1. interface that provides scalable isochronous bandwidth consumption (this is done by four alternate settings for bandwidth consumption)

A HCI frame consists of a HCI header and HCI data and it should be contained in one USB transaction. An USB transaction is defined as one or more USB frames that contain the data from one I/O request

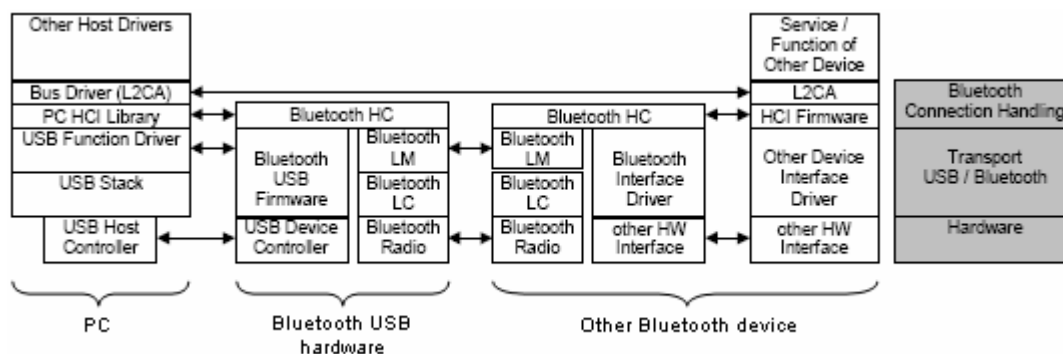


Fig. 4.6.5: Data flow from one Bluetooth device to another, using USB transport interface

Control endpoint (endpoint 0) is used to configure and control the USB device. It will also be used to allow the host to send HCI-specific commands to the Host Controller. When the USB firmware receives a packet over this endpoint that has the Bluetooth class code, it should treat the packet as an HCI command packet.

Bulk endpoints are used because of requirements for data integrity and bandwidth. By them it's possible to transfer multiple 64-byte packets per second, across the bus. It is also possible to detect and correct errors.

Interrupt endpoints are necessary for ensuring that events are delivered in a predictable and timely manner. Event packets can be sent across USB with a guaranteed latency.

Isochronous endpoints are used to transfer SCO data to and from the Host Controller of the radio. Time is critical here. The USB firmware should transfer the contents of the data to the host controller's SCO FIFOs. If the FIFOs are full, the data should be overwritten with new data.

Advantage of the USB as a transport interface is relatively high speed, but there are some power specific limitations, as well as power consumption (while a device is attached, the USB host controller continually snoops memory to see if there is any work that needs to be done).

UART transport interface

Purpose of UART transport interface layer is to make it possible to use the Bluetooth HCI over a serial interface between two UARTs on the same PCB.

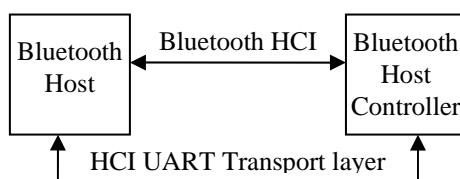


Fig. 4.6.6: HCI UART transport interface

There are four kinds of HCI packets that can be sent via the UART transport layer: HCI Command packet (it can only be sent to the Bluetooth Host Controller), HCI ACL data packet, HCI SCO data packet and HCI event packet (can be sent only from the Bluetooth Host Controller).

All four packets have a length field, so it can be determined how many bytes are expected for the HCI packet. Before sending the HCI packet, it must be sent HCI packet indicator that is used for determining the HCI packet type.

If the Host or the Host Controller lose synchronization in the communication over RS-232, then a reset is needed. A loss of synchronization means that an incorrect packet indicator has been detected, or that the length field in an HCI packet is out of range.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

4.6.5.2 Non-standard transport interface (BCSP)

Besides the standard transport interfaces, there are also transport interfaces specific for some manufacturers.

BCSP (BlueCore Serial Protocol) is transport interface created by [Cambridge Silicon Radio](#) – one of the leading companies in producing Bluetooth chips.

It is basically HCI transport layer for UART based devices, and it's implemented in the BlueCore01 and BlueCore02 chips from CSR manufacturer.

The BCSP includes checksums and retransmissions of packets which is needed for high data rates without any errors over UART lines.

4.6.6 Logical Link Control and Adaptation Protocol (L2CAP)

L2CAP adapts upper-layer protocols to the baseband layer by providing Synchronous Connection-Oriented (SCO) and Asynchronous ConnectionLess (ACL) data services to upper-layer protocols. It has protocol multiplexing capability (this means that it's possible to use more than one higher-layer protocol), segmentation and reassembly operation of data packets, and group abstractions.

It lies above the Baseband protocol (see Figure 4.6.7) and interfaces with other communication protocols as the Service Discovery Protocol (SDP), RFCOMM and Telephony Control (TCS) (packetized audio data, such as IP telephony may be used over L2CAP).

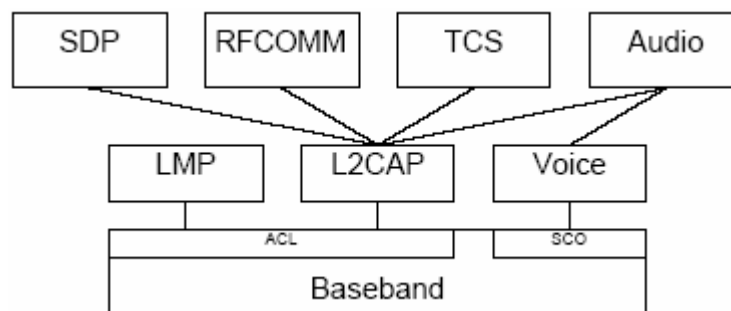


Fig. 4.6.7: L2CAP in the Bluetooth protocol architecture

Essential protocol requirements for L2CAP include simplicity and low overhead. Implementations of L2CAP must be applicable for devices with limited computational resources, as well as low power consumption.

L2CAP must support protocol multiplexing so it can be able to distinguish between upper layer protocols (SDP, RFCOMM and TCS), because the Baseband Protocol is unable to identify those services.

Data packets defined by the Baseband Protocol are limited in size, so there must be mechanism for dividing the larger packets into smaller ones, and for combining the smaller packet into larger ones (to reduce overhead data).

The L2CAP connection establishment process allows the exchange of information regarding the Quality of Service (QoS) expected between two Bluetooth units. Each L2CAP implementation must monitor the resources used by the protocol and ensure that QoS contracts are honored.

Address grouping is supported by the concept of piconets – group of devices synchronously hopping together using the same clock. The L2CAP group abstraction permits implementations to efficiently map protocol groups on to piconets. Without a group abstraction, higher level protocols would need to be exposed to the Baseband Protocol and Link Manager functionality in order to manage groups efficiently.

The Logical Link Adaptation and Control Protocol (L2CAP) is packet-based but uses channels for communications.

Each one of the end-points of an L2CAP channel is referred to by a *channel identifier* – local names representing a logical channel end-point on the device. Identifiers are in range of 0x0001 to 0x003f and they are reserved for specific L2CAP functions. Null-identifier 0x0000 represents an error and it should never be used.

Connection-oriented data channel:

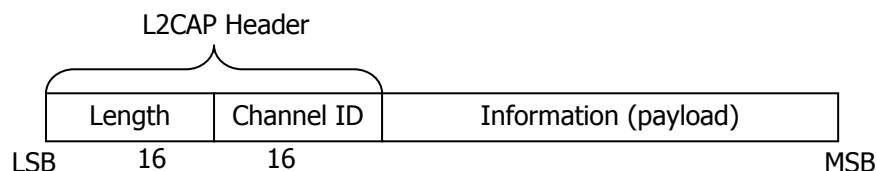


Fig. 4.6.8: L2CAP Packet - Connection-oriented channel

Length – 2 octets (16 bits) – indicates the size of the payload, excluding the length of the L2CAP header. Max size is 65536 bytes

Channel ID – 2 octets (16 bits) – identifies the channel endpoint of the packet (the scope of the channel ID is relative to the device the packet is being sent to)

Information – 0 to 65536 octets – payload received from the upper layer protocol (outgoing packet), or delivered to the upper layer protocol (incoming packet). The minimum supported MTU (Maximal Transfer Unit) for connection-oriented packets is negotiated during the channel configuration (for signaling packet the minimum is 48 bytes).

Connection-less data channel:

L2CAP protocol supports, so called group-oriented data channel. In this case, data sent to the group channel is sent to all the members of that group. Group channels are unreliable, and there is no assurance that all members of the group will get that sent data. If reliable group is required, it must be implemented at the higher layer.

Local device that sends the data to the group cannot be a member of that group, and higher level protocols are expected to loopback any data traffic being sent to the local device. Non-group members may receive group transmissions and higher level (or link level) encryption can be used to support private communication.

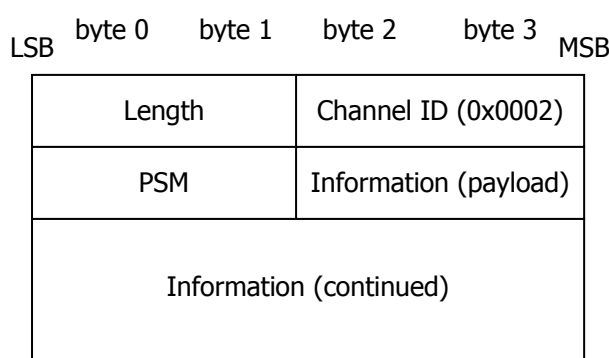


Fig. 4.6.9: L2CAP Packet – Connection-less channel

Length – 2 octets – indicates the size of information payload plus the PSM field in bytes, excluding the length of the L2CAP header

Channel ID – 2 octets – Channel ID (0x0002) reserved for connection-less traffic

Protocol/Service Multiplexer (PSM) – minimum 2 octets – this field is based on ISO 3309 extension mechanism for address fields. The least significant bit of the least significant octet of the PSM value must be '1', and also all PSM values must be assigned such that the least significant bit of the most significant object equals '0'. This allows the PSM field to be extended beyond 16 bits. PSM values are specific to L2CAP and assigned by the Bluetooth SIG.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Information – 0 to 65536 octets – the payload information to be distributed to all members of the group. Implementation must support a minimum connectionless MTU of 670 octets, unless explicitly agreed upon otherwise.

4.6.7 Service Discovery Protocol (SDP)

SDP simple protocol with minimal requirements on the underlying transport. It is used for applications to discover which services are available and to determine the characteristics of those available services on Bluetooth devices.

The service discovery mechanism provides the means for client applications to discover the existence of services provided by server applications as well as the attributes of those services. The attributes of a service include the type or class of service offered and the mechanism or protocol information needed to utilize the service.

The server maintains a list of service records that describe the characteristics of services associated with the server. Each service record contains information about a single service. A client may retrieve information from a service record maintained by the SDP server by issuing an SDP request.

A service is any entity that can provide information, performs an action, or control a resource on behalf of another entity. It may be implemented as a software, hardware or combination of software and hardware. Each service record within an SDP server is uniquely identified by the service record handle.

Each service is an instance of a service class that, by definition, provides the definitions of all attributes contained in service records that represent instance of that class. Each attribute definition specifies the numeric value of the attribute ID, the intended use of the attribute value, and the format of the attribute value. Unique identifier is assigned for each service class.

For communication is used request/response model (Fig. 4.6.10) where each transaction consists of one request protocol data unit (PDU) and one response PDU. SDP uses Big endian byte order (more-significant bytes are being transferred before less-significant bytes).

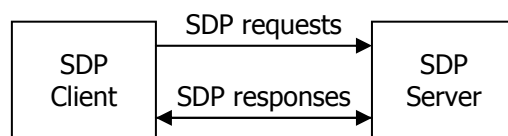


Fig. 4.6.10: Service Discovery Protocol

Each PDU consists of the PDU ID, Transaction ID, parameter length and parameters (Fig. 4.6.11.)

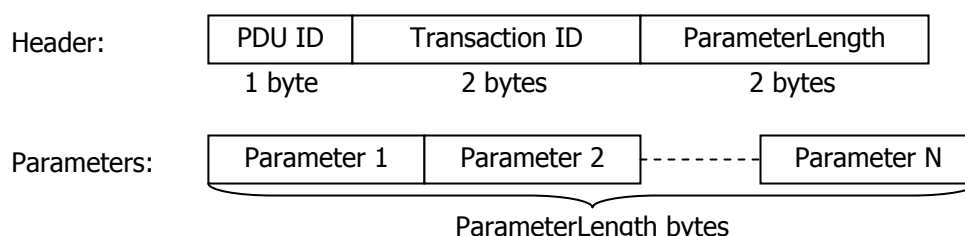


Fig. 4.6.11: Protocol Data Unit (PDU format)

Some Service Discovery Protocol requests may require responses that are larger than can fit in a single response. In this case, partial response is created along with the continuation state parameter.

In case of wrongly formatted request, or if the server is unable to respond with an

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

appropriate PRU, it may return an error_response instead of usual response.

4.6.8 Serial-port emulation protocol (RFCOMM)

The RFCOMM protocol is used as cable replacement for serial connections between devices. It provides emulation of serial ports over the L2CAP protocol. The protocol is based on ETSI standard TS07.10.

RFCOMM emulates the RS-232 serial ports, and there is even built-in scheme for null-modem emulation. Maximum simultaneous connection number is 60, but real number of simultaneous connections is implementation-specific.

RS-232 Control signals:

- 102 Signal common
- 103 Transmit Data (TD)
- 104 Received Data (RD)
- 105 Request to Send (RTS)
- 106 Clear to Send (CTS)
- 107 Data Set Ready (DSR)
- 108 Data Terminal Ready (DTR)
- 109 Data Carrier Detect (CD)
- 125 Ring Indicator (RI)

It is possible to have multiple serial ports between two devices – a Data Link Connection Identifier (DLCI) identifies an ongoing connection between a client and a server application. 6 bits represent the DLCI, but values 1 (reserved due to concept of Server Channels), 62 and 63 are unusable.

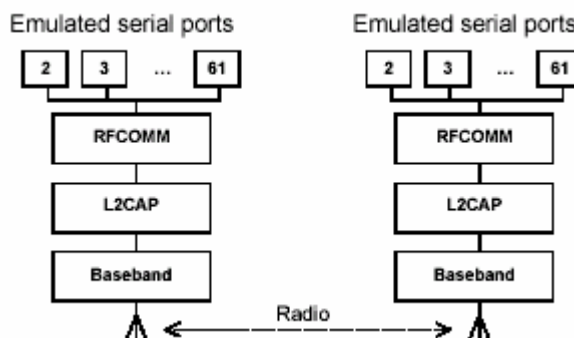


Fig. 4.6.12: Multiple emulated serial ports

Multiple emulated serial ports and multiple Bluetooth devices – if a Bluetooth device supports multiple emulated serial ports and the connections are allowed to have endpoints in different Bluetooth devices, then the RFCOMM entity must be able to run multiple multiplexer sessions.

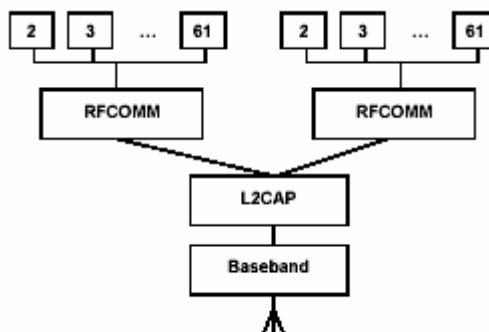


Fig. 4.6.13: Emulating serial ports coming from two Bluetooth devices

Service definition model:

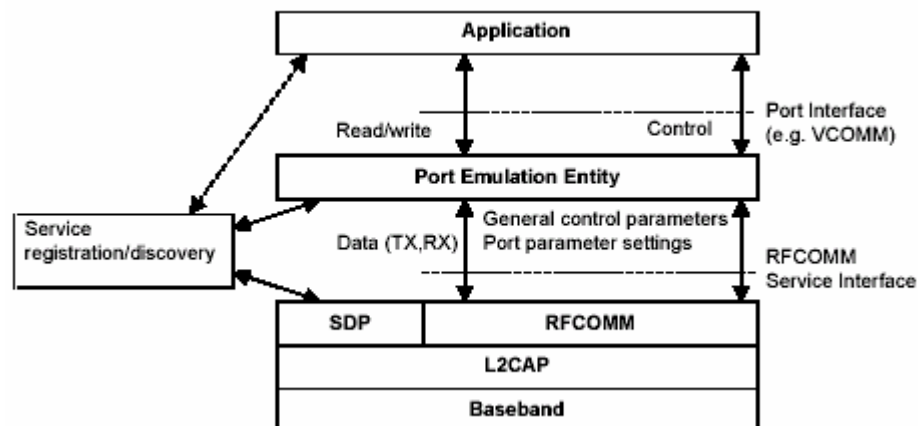


Fig. 4.6.14: RFCOMM reference model

RFCOMM layer is positioned on top of the L2CAP layer. L2CAP layer is capable of channel multiplexing, so it is possible to have multiple emulated serial ports. Server applications are registered on local device, and SDP provides services for client applications to discover how to reach server applications on other devices. RFCOMM provides a transparent data stream and control channel over an L2CAP channel.

The *Port emulation entity* maps a system-specific communication interface (API) to the RFCOMM services. In combination with RFCOMM it makes the port driver.

Applications are used for utilization of serial port communication interface.

Frame types supported in RFCOMM:

Name:	Type:
Set Asynchronous Balanced Mode (SABM)	command
Unnumbered Acknowledgement (UA)	response
Disconnected Mode (DM)	response
Disconnected (DISC)	command
Unnumbered Information with Header check (UIH)	command and response

Specification TS 07.10 defines a multiplexer that has dedicated control channel, DLCI 0. It is used to convey information between two multiplexers.

Supported Control Channel Commands:

- Test Command (Test)
- Flow Control On Command (FCon)
- Flow Control Off Command (FCoff)
- Modem Status Command (MSC)
- Remote Port Negotiation Command (RPNC)
- Remote Line Status (RLS)
- DLC parameter negotiation (PN)
- Non Supported Command response (NSC)

Start-up procedure of serial port communication over RFCOMM:

- Establish an L2CAP channel to the peer RFCOMM entity, using L2CAP service primitives
- Start the RFCOMM multiplexer by sending SABM (Set Asynchronous Balanced Mode) command on DLCI 0 and await UA response from peer entity.
- Data Link Connections (DLC) can be now established for user data traffic.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Closedown procedure of serial port communication over RFCOMM:

- The Device closing the last connection (DLC) on a particular session is responsible for closing the multiplexer by closing the corresponding L2CAP channel. Closing the multiplexer by first sending a DISC command frame on DLCI 0 is optional, but it is mandatory to respond correctly to DISC (with UA response).

Link-loss handling:

If an L2CAP link loss notification is received, the local RFCOMM entity is responsible for sending a connection loss notification to the port emulation/proxy entity for each active DLC.

Flow control in the wired systems is based on RTS/CTS signals, but the flow control between RFCOMM and the lower layer L2CAP depends on the service interface supported by the implementation. RFCOMM has also its own flow control mechanisms, and L2CAP relies on the flow control mechanism provided by Link Manager layer in the Baseband.

Interaction with other entities:

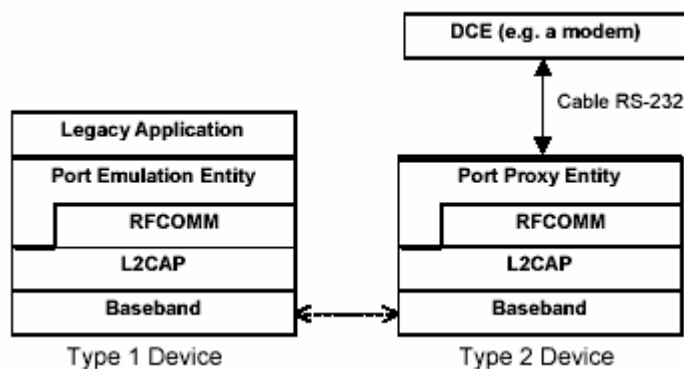


Fig. 4.6.15: The RFCOMM communication model

Type 1 devices are communication endpoints such as computers and printers.

Type 2 devices are part of a communication segment (i.e. modems).

The *Port Emulation Entity* maps a system specific communication interface (API) to the RFCOMM services.

The *Port Proxy Entity* relays data from RFCOMM to the external RS-232 interface linked to a DCE (Data Circuit-Terminating Equipment) – communication parameters are set according to received RPN commands.

4.6.9 Object EXchange protocol (OBEX)

OBEX is protocol that Bluetooth adopted from IrDA. Original name of this protocol (defined by IrDA) is IrOBEX, but shorter name OBEX is used instead. By adopting OBEX, it is possible to use it over IrDA IR (light) or Bluetooth technology (radio signals).

OBEX was developed to exchange data objects over an infrared link and was placed within the IrDA protocol hierarchy. However, it can appear above other transport layers, now RFCOMM and TCP/IP. At this moment, it is worth mentioning that the OBEX over TCP/IP implementation is an optional feature for Bluetooth applications supporting the OBEX protocol.

The IrOBEX protocol follows a client/server request-response paradigm for the conversation format.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

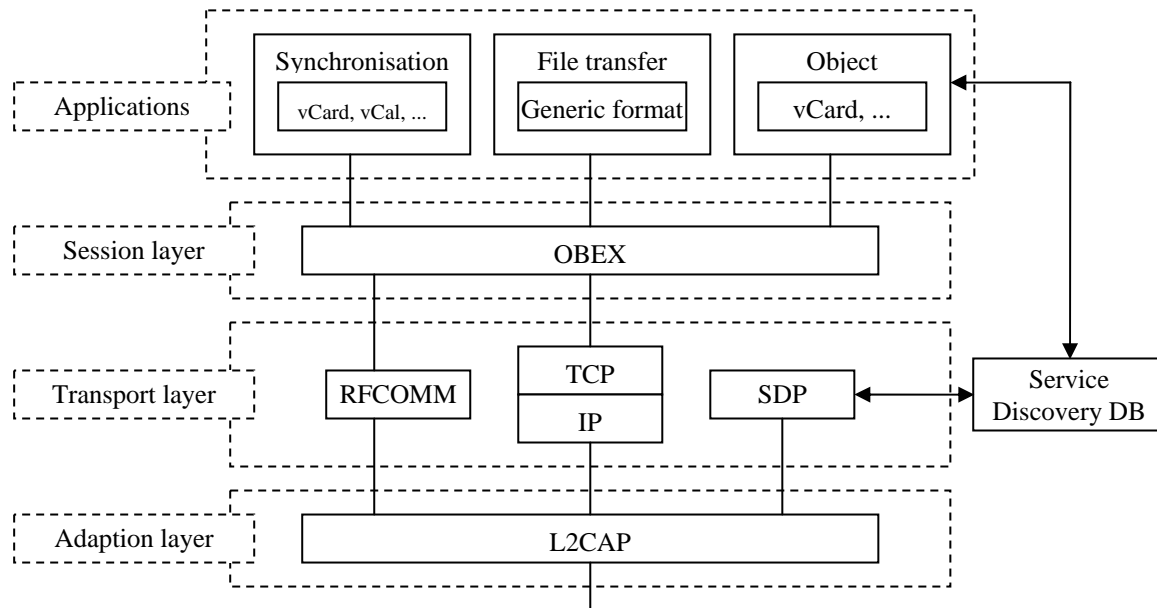


Fig. 4.6.16: OBEX hierarchy diagram

In the Bluetooth system, the purpose of the OBEX protocol is to enable the exchange of data objects. The typical example could be an object push of business cards to someone else. A more complex example is synchronizing calendars on multiple devices using OBEX.

For the Object Push and Synchronization applications, content formats can be the vCard, vCalendar, vMessage, and vNotes. Those content formats describe the formats for the electronic business card, the electronic calendar and scheduling, the electronic messages and mails, and the electronic notes, respectively.

OBEX protocol can transfer an object by using the **Put** and **Get** operations. One object can be exchanged in one or more Put requests or Get responses. The model handles both information about the object (e.g. type) and object itself. It is composed of headers, which consist of a header ID and value. The header ID describes what the header contains and how it is formatted, and the header value consists of one or more bytes in the format and meaning specified by header ID.

The specified headers are: Count, Name, Type, Length, Time, Description, Target, HTTP, Body, End of Body, Who, Connection ID, Application Parameters, Authenticate Challenge, Authenticate Response, Object Class, and User-Defined Headers.

Session protocol – the OBEX operations are formed by response-request pairs. Requests are issued by the client and responses by the server. After sending a request, the client waits for a response from server before issuing a new request.

Connect operation:

- An OBEX client starts the establishment of an OBEX connection.
- At the remote host, the OBEX server receives the Connect-request, if it exists. The server accepts the connection by sending the successful response to the client.

Disconnect operation:

- The disconnection of OBEX session occurs when an application, which is needed for an OBEX connection, is closed or the application wants to change the host to which the requests are issued. The client issues the Disconnect-request.

Put operation:

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

- When the connection has been established between the client and server the client is able to push OBEX objects to the server.

A Put-request consists of one or more request packets, depending on how large the transferred object is, and how large the packet size is. A response packet from the server is required for every Put-request packet.

Get operation:

- When the connection has been established between the client and server, the client is also able to pull OBEX objects from server.

SetPath operations:

- set path operation is used for setting the path on the remote device. This is sometimes used before put or get operations

Abort operation:

- while transferring larger objects, for which is needed more than one put or get request, sometimes is needed to abort the transfer. In that case, transmission of that large object is aborted, and there is no more sending or receiving data of that object

OBEX over RFCOMM – The Bluetooth devices supporting the OBEX protocol must satisfy the following requirements:

- The device supporting OBEX must be able to function as a client, a server, or both.
- All servers running simultaneously on a device must use separate RFCOMM server channels.
- Applications (service/server) using OBEX must be able to register the proper information into the service discovery database. This information for different application profiles is specified in the profile specifications.

OBEX server start-up on RFCOMM:

- When a client sends a connecting request, a server is assumed to be ready to receive requests. However, before the server is ready to receive (i.e. is running) certain prerequisites must be fulfilled before the server can enter the listening mode:
 - The server must open an RFCOMM server channel
 - The server must register its capabilities into the service discovery database

Connection establishment:

- A client initiates the establishment of a connection. However, the following sequence of tasks must occur before the client is able to send the first request for data:
 - o By using the SD protocol described in SDP specification, the client must discover the proper information (e.g. RFCOMM channel) associated with the server on which the connection can be established.
 - o The client uses the discovered RFCOMM channel to establish the RFCOMM connection
 - o The client sends the Connect-request to the server, to establish an OBEX session. The session is established correctly if the client receives a successful response from the server.

Disconnection:

- Using the Disconnect-request does the disconnection. When the client has received the response, the next operation is to close the RFCOMM channel assigned to the OBEX

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

client.

4.7 Bluetooth security

Bluetooth devices are used for wireless communication. Wireless means that data is transferred through air as a medium. In order to prevent third party from getting confidential Information (that is sent as a plain text) by *sniffing* traffic it's good idea to encrypt that data. By that way, even if the third person gets to the data, it won't be able to decipher it.

Security in Bluetooth goes through many layers: SAFER+ algorithm is implemented for security procedures at the baseband layer, Link Manager specification covers procedures for link security settings, Host Controller Interface provides interface for Host to control security settings of device, Generic Access Profile defines security modes and link, channel, connection establishment procedures between devices working in mentioned security modes.

In this implementation of Bluetooth stack most of security concerns will be considered in the Bluetooth Control Center.

According to JSR-82 specification Bluetooth Control Center (BCC) should provide:

- base security settings of the device, including the security modes defined in the Bluetooth specification
- list of remote Bluetooth devices (in vicinity) that are already known to the local device
- list of remote Bluetooth devices (in vicinity) that are trusted by local device
- mechanism for providing authorization on connection requests

4.7.1 Authentication

The authentication is based on challenge – response scheme that uses symmetric secret key. The verifier sends an RAND A which contains random number (challenge) to claimant. The claimant calculates the response, which is the function of the challenge, claimant BD_ADDR and a secret key. The response is sent back to verifier, which checks if the response was correct or not. The response will be valid only if claimant knows the secret key. Secret key is link key obtained from pairing process. Pairing process is initiated when devices are communicating for the first time and one (or both) asks for authentication. During pairing mutual authentication is done and special key, link key is generated and stored on both devices. Figure 4.7.1 shows authentication process.

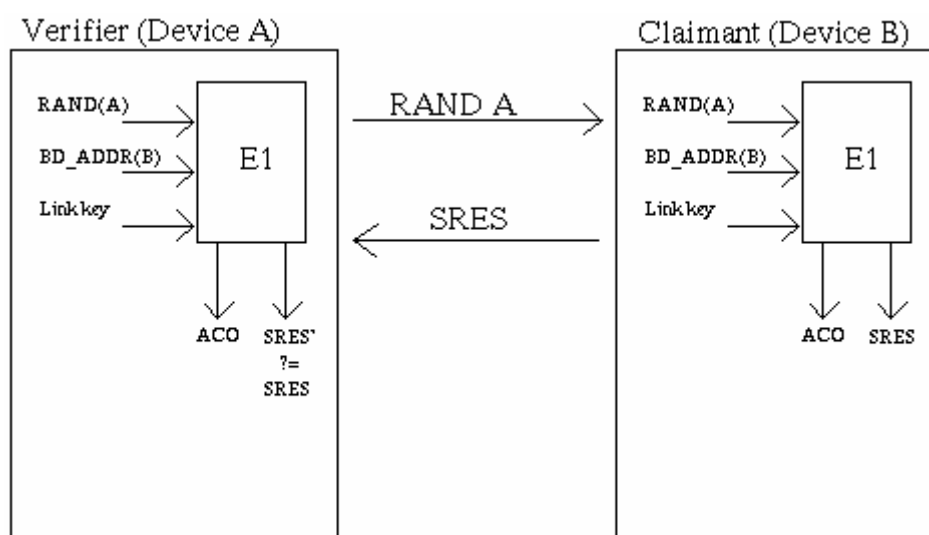


Fig. 4.7.1: Authentication procedure

SRES is calculated by E1 algorithm using generated random number, claimants address and secret link key. This value verifier later compares to one returned by claimant (SRES'). If they're equal, remote device is authenticated and ACO (Authenticated Ciphering Offset) is stored so it could

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

be used later in encryption key generation process.

The authentication function E1 uses the encryption function called SAFER+. More on SAFER+ encryption function can be found at Bluetooth Specification v1.1, page 170 (see Books, [2]).

Link Manager is responsible for controlling authentication process. When Link Manager activates authentication, it does so by sending LMP_au_rand PDU (Protocol Data Unit). Claimant responds with LMP_sres as shown in Figure 4.7.2

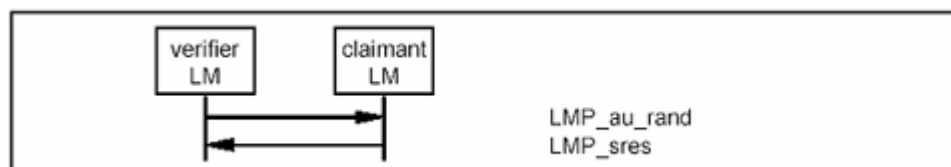


Fig. 4.7.2: Authentication

Authentication algorithm is built in. It's at the user to decide to apply it or not. Controlling Link Manager is possible through Host Controller Interface which defines commands that user can use to change device's behavior. Among many commands some of them serve to change authentication settings for device.

Command Write_Authentication_Enable based on parameter given makes that every device that is trying to connect to local device must pass authentication process before link setup is complete.

Command Authentication_Requested enabled device to ask remote device to authenticate himself on already established link.

For more detailed information about this (and other) commands mentioned above see Bluetooth Specification v1.1, page 537 (see Books, [2]). By the General Access Profile (GAP), every device should respond to authentication request, no matter on selected security mode.

Encryption and authentication are hardware issues and every Bluetooth module needs to implement them.

4.7.2 Encryption

User information can be protected by encryption of the packet payload. The access code and the packet header are never encrypted, but the payloads is carried out with a stream cipher called E0 that is re-synchronized for every payload. The overall principle is shown in Figure 4.7.3.

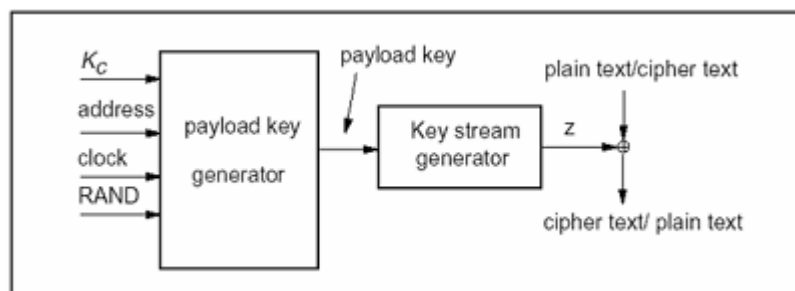


Fig. 4.7.3: Stream ciphering for Bluetooth with E0

The stream cipher system E0 consists of three parts.

- part that performs the initialization (generation of the payload key)
- part that generates the key stream bits
- part that performs the encryption and decryption.

The payload key generator is very simple - it merely combines the input bits in an appropriate order and shift them into the four LFSRs used in the key stream generator. The main part of the

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

cipher system is the part that generates the key stream bits, as it also will be used for the initialization.

The key stream bits are generated by a method derived from the summation stream cipher generator attributable to Massey and Rueppel. The method has been thoroughly investigated, and there exist good estimates of its strength with respect to presently known methods for cryptanalysis. Although the summation generator has weaknesses that can be used in so-called correlation attacks, the high re-synchronization frequency will disrupt such attacks.

Each packet payload is ciphered separately. The cipher algorithm uses the master Bluetooth address, 26 bits of the master real-time clock (CLK_{26-1}) and the encryption key as input, see Figure 4.7.4 (where it is assumed that unit A is the master).

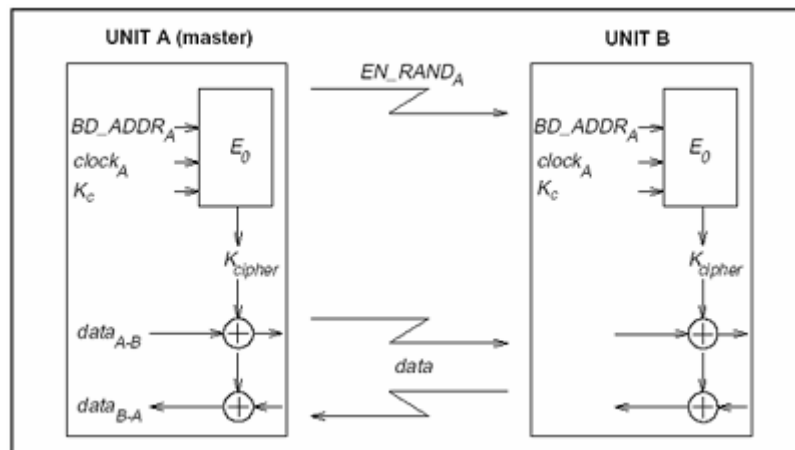


Fig. 4.7.4: Functional description of encryption procedure

The encryption key K_C is derived from the current link key, COF (Chipering Offset), and a random number (issued by the master before entering encryption mode), EN_RAND_A (see Figure 4.7.5). If current link key is master link key (used for broadcast encrypted messages) then it's function of BD_ADDR of that device, else COF equals ACO (Authenticated Chipering Offset).

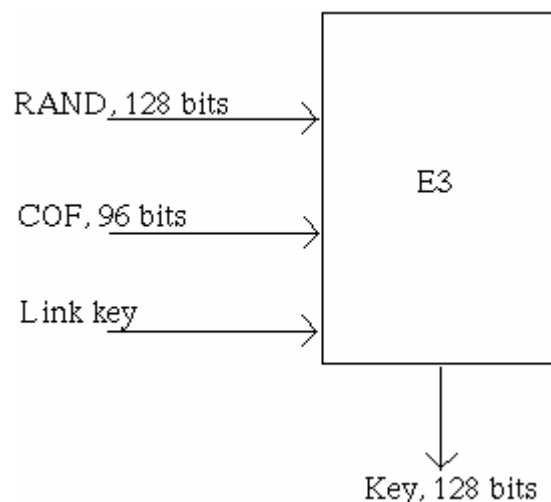


Fig. 4.7.5: Generating encryption key using E3

Notice: EN_RAND_A is publicly known since it is transmitted as plain text over the air. Within the E_0 algorithm, the encryption key K_C is modified into another key denoted K'_C . The maximum effective size of this key is factory preset and may be set to any multiple of eight between one and sixteen (8-128 bits).

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

The real-time clock is incremented for each slot. The algorithm is re-initialized at the start of each new packet. By using CLK_{26-1} at least one bit is changed between two transmissions. Thus, a new keystream is generated after each reinitialization. For packets covering more than a single slot, the Bluetooth clock as found in the first slot is being used for the entire packet.

The encryption algorithm E0 generates a binary keystream, K_{cipher} , which is modulo-2 added to the data to be encrypted, and the cipher is symmetric (decryption is the same as encryption – even the keys are the same).

5. Java Bluetooth Stack – package overview

5.1 hr.fer.rasip.bluesec package (BCC – Bluetooth Control Center)

The BCC (Bluetooth Control Center) is responsible for security and it provides methods for authentication requests. Purpose of the `hr.fer.rasip.bluesec` package is to implement BCC in the Java Bluetooth Stack. Figure 5.1.1 shows the UML class diagram of this package and its relations to the other packages.

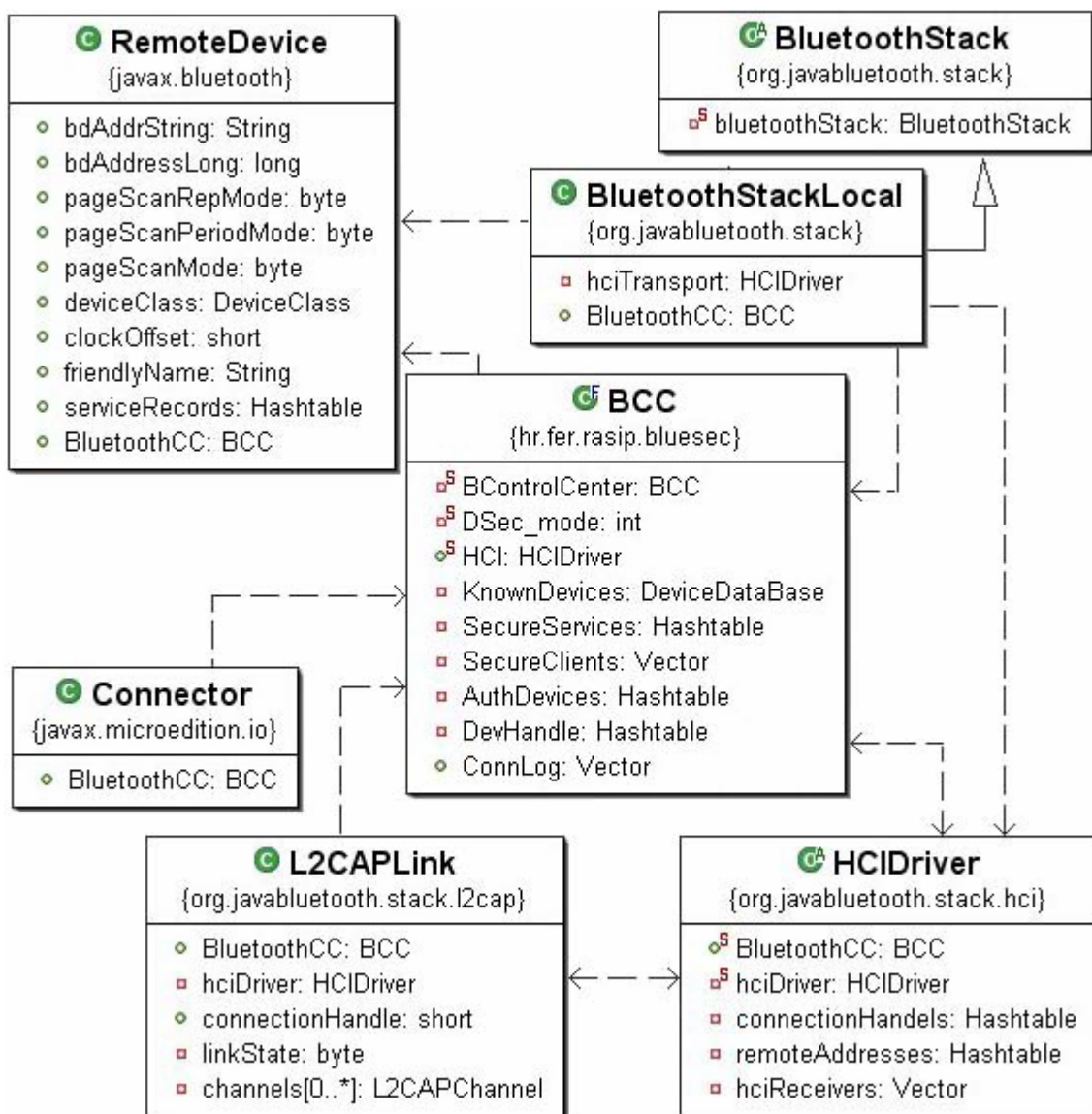


Fig. 5.1.1: class diagram for implementation of Bluetooth security methods

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

In order to control authentication settings of local device commands `Write_Authentication_Enable` and `Authentication_Requested` needs to be implemented. Implementation of those methods is in `HCIDriver` class.

Method `BCC.Authentication_Request(short conn_handle)` sends authentication request to remote device identified by `conn_handle`. `Conn_handle` is result of established connection between two devices, and it's unique for every connection.

If device is already authenticated then there is no need to go through authentication process again. Database of authenticated devices is checked to see whether the device is authenticated, and if it's authenticated then method returns true, otherwise the method `HCIDriver.send_HCI_Authentication_Requested(conn_handle)` is invoked.

`HCIDriver.send_HCI_Authentication_Requested(short)` is implementation of `Authentication_Requested` command.

When *Authentication_Requested* is invoked Host sends command packet to Host Controller. This is done with method `send_HCI_Command_Packet(byte[] data)`.

`Send_HCI_Command_Packet(byte[] data)` has inner loop that waits for Host Controller's response to the executed command. Response is indicated by *Command Complete* or *Command Status* Event (depends on command). Byte array `data` contains information about type of packet, operation code, number of bytes, connection handle as array of bytes. After that, the packet is being sent. Upon receiving the packet Host Controller will check the packet and execute the command if can. *Command Status* Event is then generated.

When the response is received, `send_HCI_Authentication_Requested` method will return it. It takes some time for authentication process to be finished. When finished *Authentication Complete* Event is received, and Host sends information attached to that Event to BCC using `authentication_report(byte[] data)` method call.

The `authentication_report` method will set executed flag that signals `BCC.Authentication_Request(short)` that authentication process is done and waiting is over. If executed flag (status report) is zero then authentication process succeeded and method returns true, else *Authentication* process has failed and method returns false. Every failure needs reason. Value in executed flag identifies that reason according to Bluetooth specification.

This implementation of BCC contains two `Authentication_Request` methods. Both take just one parameter. One seen above takes short (connection handle) while other takes instance of `RemoteDevice` class. As said before connection handle identifies connection between devices, so remote device identifies connection handle.

BCC contains reference to database that holds information about remote devices still connected to local device and connection handles that identify those connections.

`BCC.Authentication_Request(RemoteDevice)` is implemented in that way that it gets connection handle from database and calls earlier explained method. Database that contains information about authentication status of device as well as one that keeps information about remote devices and handles are implemented as instances of *Hashtable* class.

Asking whether authentication has been set before link's setup is done by using `BCC.Set_authentication_policy_on_link_level(byte enable)` method call. Parameter `enable` if set to one enables that authentication will be required before Link setup phase is complete. Parameter set to zero disables that option. When invoked, method `HCI.send_HCI_Write_Authentication_Enable(byte enable)` sends message to the `HCIDriver`.

`Send_HCI_Write_Authentication_Enable(byte enable)` is implementation of *Write_Authentication_Enable* command. It sends Host Controller command Packet containing data about what kind of packet is it, which command, how many bytes it contains and enable parameter.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

When Host Controller receives that packet and executes specified action, Command Complete Event is raised.

When Host receives that event, it checks the status. If status equals zero then `send_HCI_Write_Authentication_Enable` method returns true, else it returns false. Value returned by `send_HCI_Write_Authentication_Enable` is value returned by `Set_authentication_policy_on_link_level`.

All `HCIDriver` method calls to Host Controller could result with exception `HCIException` as consequence of I/O error.

Sequence diagram for authentication procedure is shown on Figure 5.1.2

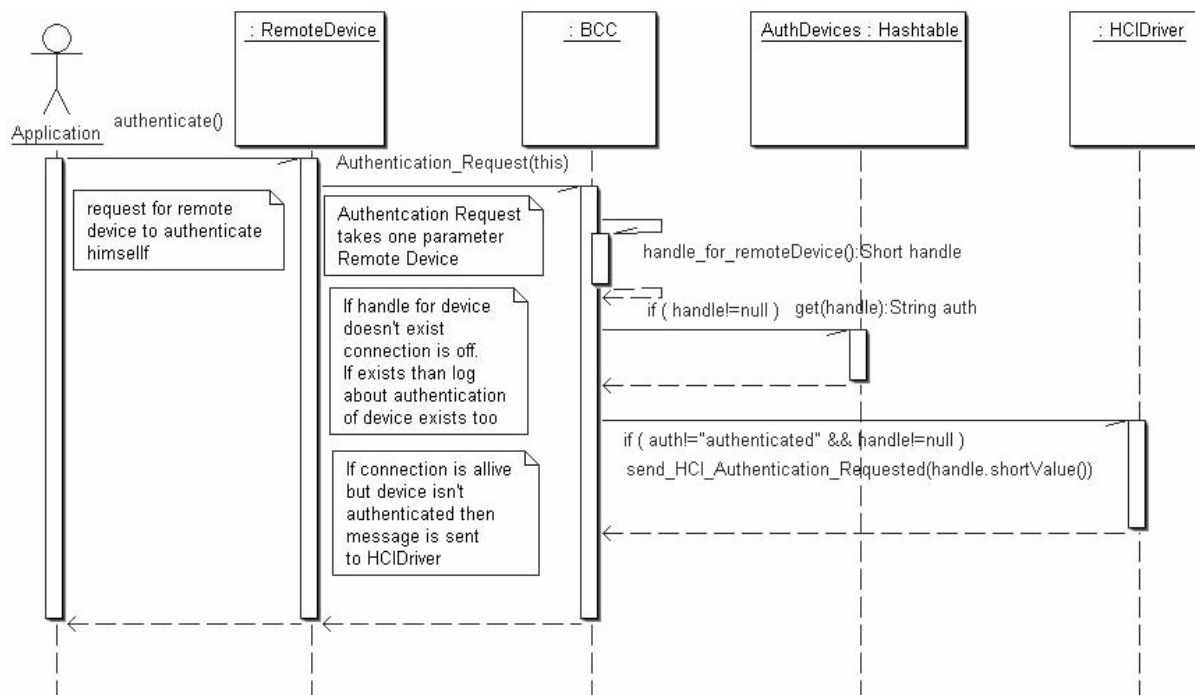


Fig. 5.1.2: Sequence diagram for authentication procedure

Commands for enabling and disabling encryption are also available. Host Controller Interface defines those commands and they are implemented in `HCIDriver` class.

Setting encryption on, over established link is done using `BCC.Encryption_Request(short conn_handle, byte on)` method. Parameter `conn_handle` identifies connection, while parameter `on` specifies should connection be encrypted or not. Suppose that local device is using two or more services of remote device. If one of those services requires encryption then encryption will be applied on all connections. That isn't written in Bluetooth specification, but it's one of possibilities mentioned in JSR82 specification that's left to implementation of BCC.

`Encryption_Request` method is called in 3 cases:

- on the client side when the connection is being established (`clientSecurityProcedures()` method)
- on the server side when the connection is being established (`serverSecurityProcedures()` method)
- when `RemoteDevice.encrypt()` method is invoked

If device security mode is 3 then encryption is on from link setup between devices and you

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

can't shut it down. Attempt to shut down encryption (on = 0x00) if device in mode 3 will result with method to return with false. If an attempt is started to turn encryption on (on = 0x01) when device in mode 3 return value is true.

If device isn't in mode 3 BCC sends `send_HCI_Set_Connection_Encryption(conn_handle,enable)` to HCIDriver. Upon receiving message HCIDriver forms packet containing information about packet type, operation code, length of packet data, data: `conn_handle(byte array)` and enable. Then issues `send_HCI_Command_Packet(byte[] data)` with explained data as parameter.

`Send_HCI_Command_Packet` will wait for response. When packet comes to Host Controller it process it and raises *Command Status Event*. That's response that `send_HCI_Command_Packet` was waiting for. `Send_HCI_Set_Connection_Encryption` is now done and program flow is in waiting loop. Waiting for *Encryption Change Event* to be come. When Encryption Change Event is received data attached to him is sent to BCC where status flag (named executed) is set. If status flag is zero then method returns true, else it's false.

Methods `authentication_report(byte[] data)` and `encryption_report(byte[] data)` are some sort of messages that set status flag.

Methods `isAuthenticated(RemoteDevice)`, `isEncrypted(RemoteDevice)`, `isTrusted (RemoteDevice)` are implemented in same way. First connection handle is obtained from the database that contains information about connection handles and remote devices that use them. In source, this database is referenced as `DevHandle`. Now when connection handle is known other databases are searched for records containing that connection handle.

Method `isAuthenticated(RemoteDevice)` searches through *AuthDevices* database, `isEncrypted(RemoteDevice)` searches *ConnLog* database and `isTrusted(RemoteDevice)` searches database.

During connection establishment, when calling `Connector.open(stringURL)` method certain security requests are contained in stringURL. In order to apply those requirements, they are stored in database.

There are two databases. One for client connection requirements and one for service connection requirements. After parsing stringURL in `Connector.open(String)` method new records are gathered and sent to BCC. Service requirements are gathered using `BCC.createServiceReqRecord(psm, authenticate, authorize, encrypt)` method. Client requirements are gathered using `BCC.channelSecurityRequirement(remote_bdaddr, psm, authenticate, encrypt)` method. Every method adds new record to database. Records differ.

New client channel record contains information about remote device address, psm identifying service and authenticate, authorize, encrypt parameters. In this case authorize is always false because there's no logic in client for doing authorization.

Authenticate, authorize and encrypt instantiates class `SecRequirement` that is used by service database, too. Client database is named *SecureClients* and implemented as *Vector*.

New service record is instance of class `SecRequirement`. This time authorize parameter makes sense and is not always false. Service database is named *SecureServices* and is implemented as *Hashtable*. Because it's *Hashtable* psm serves as a key for extracting records.

Method `client_security_procedures(long r_addr, short psm)` tries to apply Client requirements specified in *SecureClients* database records. Parameters `r_addr` and `psm` are used to retrieve relevant record. When record is extracted and connection parameters are red form *SecRequirement* part of the record then security procedures are called. That means if authenticate is true `BCC.Authentication_Request` is invoked, and if encrypt is true

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

`BCC.Encryption_Request` is called. In order for method to return true all invoked methods must pass. That means if authentication fails - all fails, if encryption fails - all fails. There is one exception when `encrypt` parameter is false but other connections are using encryption over the same connection handle then method will return true no matter parameter says false. That is taken care off by using function `current_count(short conn_handle)` which returns number of encrypted connections over connection handle retrieved using `r_addr` parameter. If that number is higher than zero method returns true. So when negative request comes if `count > 0` return value is true.

Method `service_security_procedures(short psm, short conn_handle)` is very similar. Record is extracted from *SecureServices* database using `psm` as key. Everything else is the same except there's another security procedure, `Authorisation_Request(conn_handle)` that retrieves record identified by `conn_handle` from *KnownDevices* database. If records authorize parameter is "trusted" device has access to all services.

KnownDevices database keeps track of all devices in the neighborhood.

Generic Access Profile defines 3 security modes in which device can work in:

- **Security mode 1 (non-secure):** When a Bluetooth device is in security mode 1 it shall never initiate any security procedure (i.e., it shall never send `LMP_au_rand`, `LMP_in_rand` or `LMP_encryption_mode_req`).
- **Security mode 2 (service level enforced security):** When a Bluetooth device is in security mode 2 it shall not initiate any security procedure before a channel establishment request (`L2CAP_ConnectReq`) has been received or a channel establishment procedure has been initiated by itself. Whether a security procedure is initiated or not depends on the security requirements of the requested channel or service.

A Bluetooth device in security mode 2 should classify the security requirements of its services using at least the following attributes:

- Authorization required
- Authentication required
- Encryption required

According to the security mode 2 `service_security_procedures` are invoked when *L2CAP_ConnReq Event* has come (device working as server) or when *L2CAP_ConnReq* has been sent (client, `client_security_procedures` which is placed in `L2CAPLink.connect_l2cap_channel` method call after `send_l2cap_connection_req`)

- **Security modes 3 (link level enforced security):** When a Bluetooth device is in security mode 3 it shall initiate security procedures before it sends `LMP_link_setup_complete`. A Bluetooth device in security mode 3 may reject the host connection request (respond with *LMP_not_accepted* to the `LMP_host_connection_req`) based on settings in the host (e.g. only communication with pre-paired devices allowed). In this implementation host in mode 3 will reject connection request if device isn't trusted

Sequence diagram for encrypted connection is shown on the figure 5.1.3, and the sequence diagram for the overall client security is shown on figure 5.1.4.

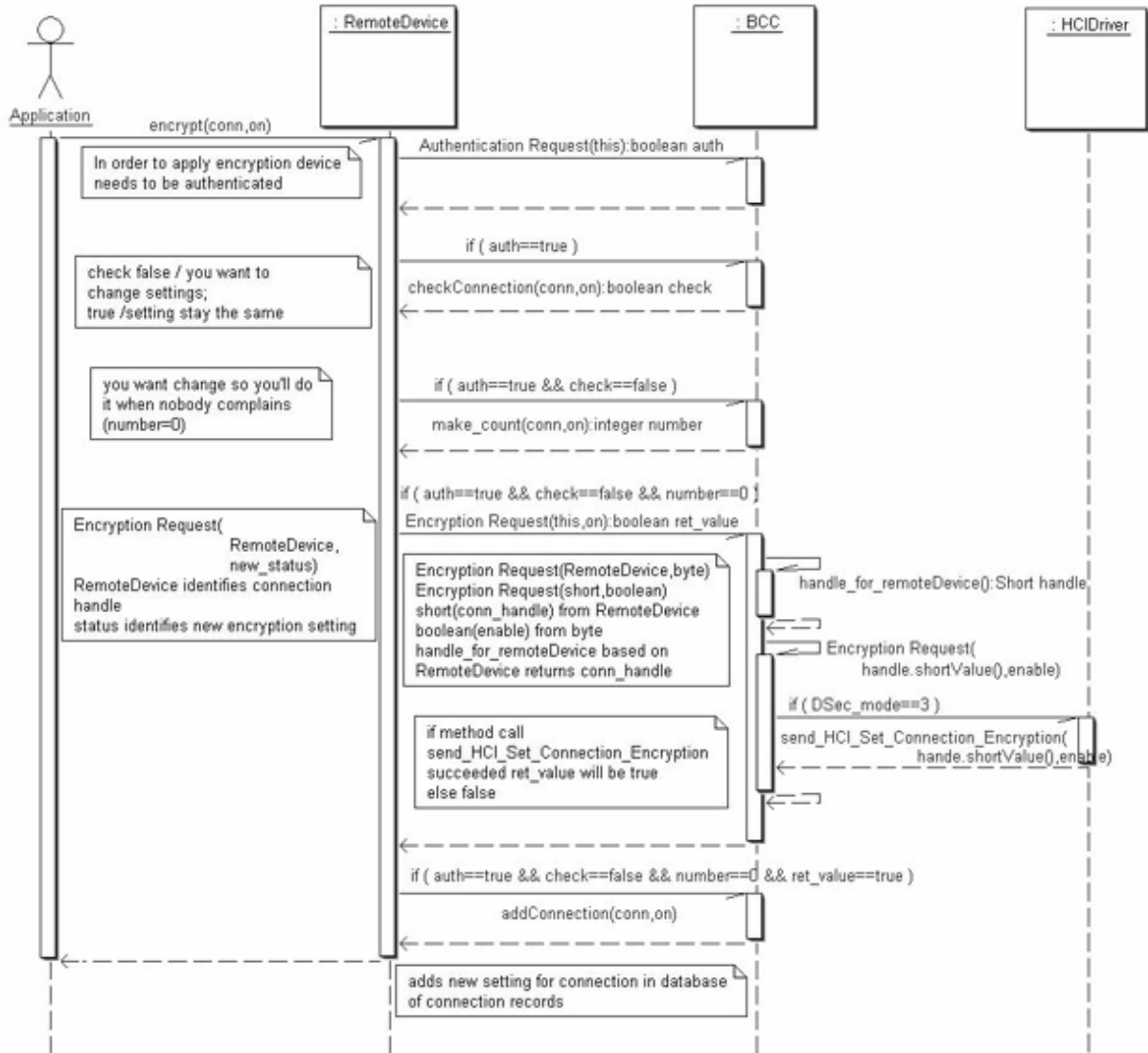


Fig. 5.1.3: BCC - encrypted connection sequence diagram

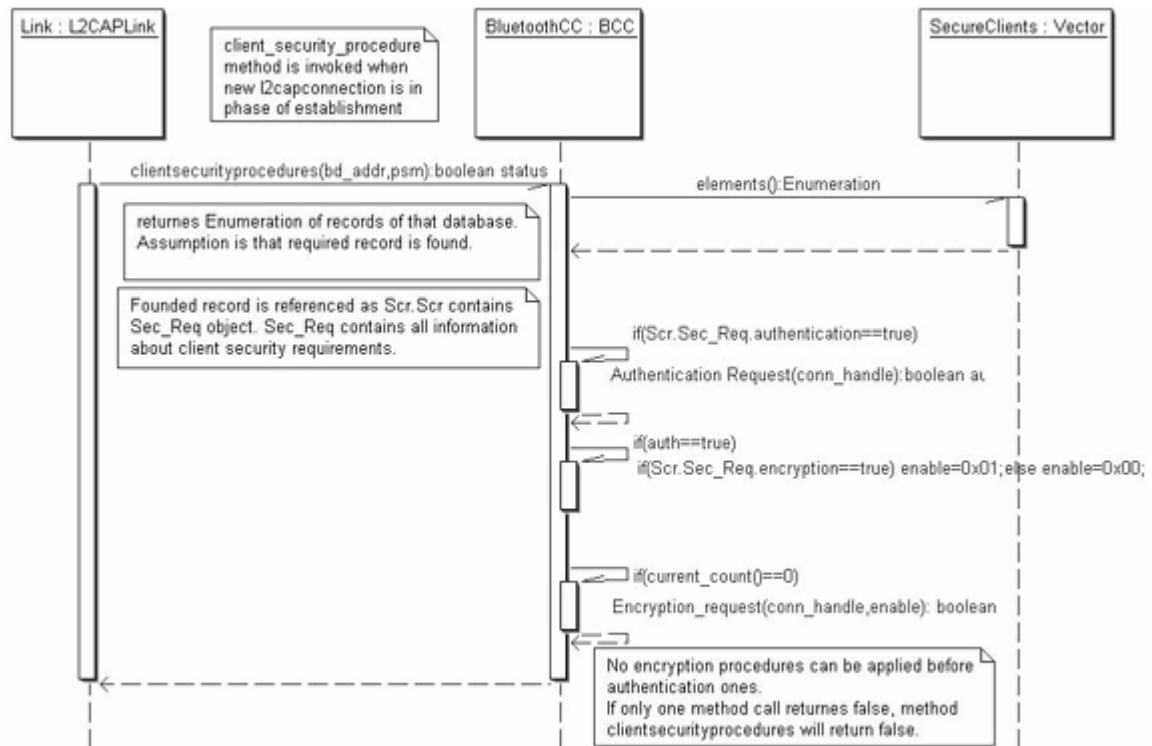


Fig. 5.1.4: BCC - Client security

5.2 hr.fer.rasip.rfcomm package

The RFCOMM protocol provides emulation of serial ports over the L2CAP protocol. It supports up to 60 simultaneous connections between two Bluetooth devices. Concrete number of connections is implementation specific. The protocol is based on the ETSI standard TS 07.10.

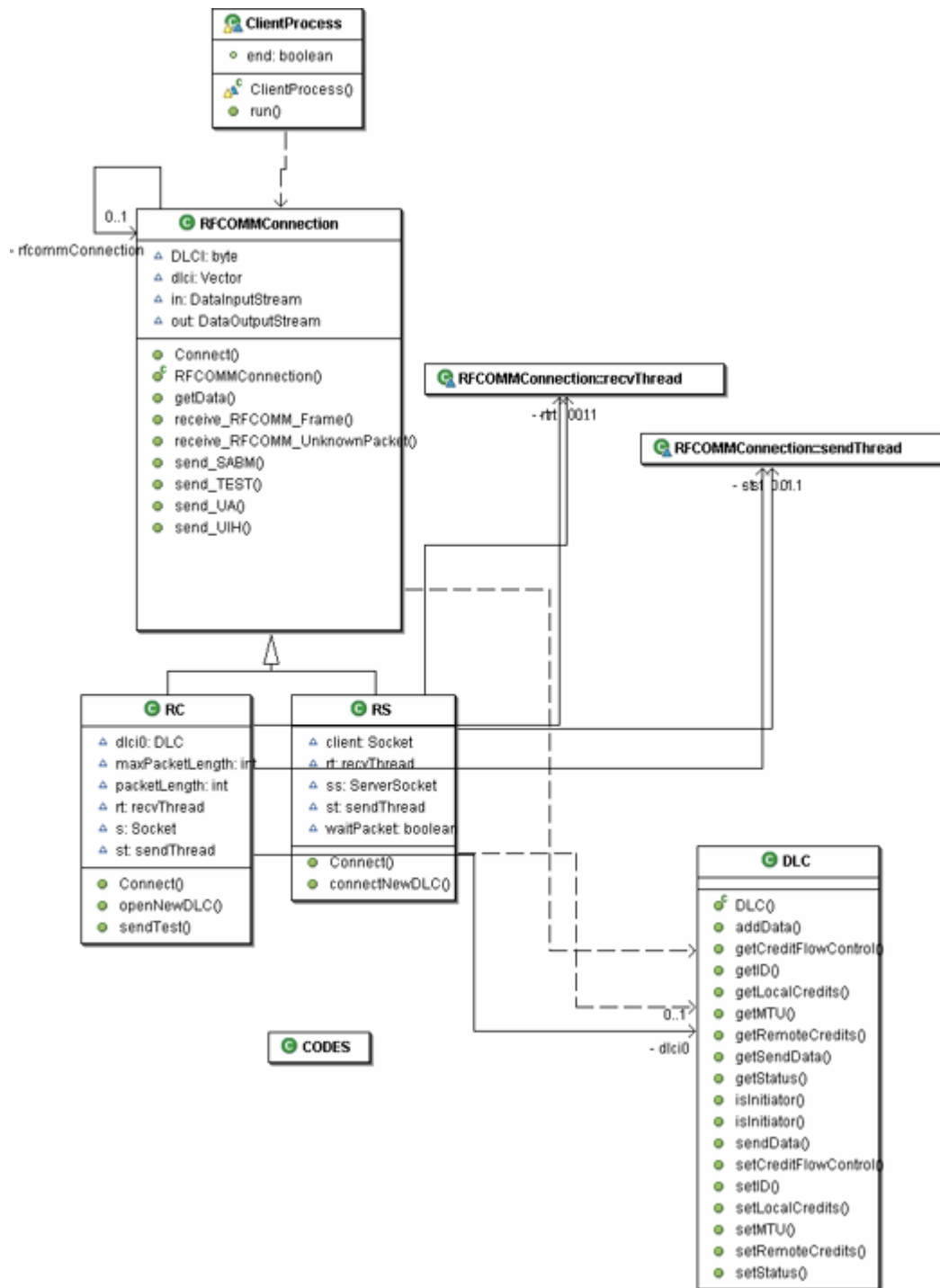


Fig. 5.2.1: RFCOMM class diagram

With each connection opened a new DLC object is created on both sides (client and server). The DLC object contains all the information regarding a specific channel, like: local credits, remote credits, status, mtu, ID (DLCI), information about the initiating device, credit flow control support on current channel and an input and output buffers.

All these channel options are made private and can be accessed by appropriate functions:

```

setID(),          getID(),          setRemoteCredits(int),      getRemoteCredits(),
setCreditFlowControl(boolean),      getCreditRemoteControl(),
setLocalCredits(int),  getLocalCredits(),  setStatus(int),  getStatus(),

```

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

setMTU(int), getMTU(), isInitiator(boolean) and isInitiator().

Also, there are functions for reading and writing data to input and output buffer: sendData(byte[]), getSendData(byte[]), addData(byte[]) and getRecvData(byte[]).

There are 5 supported frame types in RFCOMM:

1. Set Asynchronous Balanced Mode (SABM) command
2. Unnumbered Acknowledgement (UA) response
3. Disconnected Mode (DM) response
4. Disconnected (DISC) command
5. Unnumbered information with header check (UIH) command and response

There are 8 supported control commands in RFCOMM:

1. Test Command (test)
2. Flow Control On Command (FCon)
3. Flow Control Off Command (FCoff)
4. Modem Status Command (MSC)
5. Remote Port Negotiation Command (RPN)
6. Remote Line Status (RLS)
7. DLC Parameter Negotiation (PN)
8. Non Supported Command Response (NSC)

Each L2CAP frame always contains exactly one RFCOMM frame. When a new L2CAP frame is received it is automatically parsed in receive_RFCOMM_Frame(byte[]) method of the RFCOMMConnection object.

Flag	Address	Control	Length Indicator	Information	FCS	Flag
0111 1101	1 octet	1 octet	1 or 2 octets	Unspecified length but integral number of octets	1 octet	0111 1101

Fig. 5.2.2: The format of each RFCOMM frame

Address field from TS 07.10. specification is redefined in RFCOMM specification (Fig. 5.8.3)

Bit No.	1	2	3	4	5	6	7	8
TS 07.10	EA	C/R	DLCI					
RFCOMM	EA	C/R	D	Server Channel				

Fig. 5.2.3: RFCOMM address field

In receive_RFCOMM_Frame(byte[]) first of all EA, C/R and D bits are extracted, and DLCI numbers are transformed into integer.

Frame Type	1	2	3	4	5	6	7	8	Notes
SABM (Set Asynchronous Balanced Mode)	1	1	1	1	P/F	1	0	0	
UA (Unnumbered Acknowledgement)	1	1	0	0	P/F	1	1	0	
DM (Disconnected Mode)	1	1	1	1	P/F	0	0	0	
DISC (Disconnect)	1	1	0	0	P/F	0	1	0	
UIH (Unnumbered Information with Header check)	1	1	1	1	P/F	1	1	1	
UI (Unnumbered Information)	1	1	0	0	P/F	0	0	0	Optional

Fig. 5.2.4: Control field

P/F bit is cleared before determining the exact frame type.

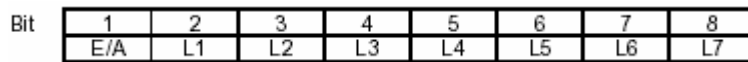


Fig. 5.2.5: Length field

Length field can be 2 bytes long if the E/A bit is set. The length of the data is calculated using the method `getLength(byte[], int)`

FCS is Frame Checking Sequence Field, and it is created/checked by two methods in `RFCOMMConnection`: `get_CRC(byte[], int)` and `check_CRC(byte[], int, byte)`.

The appropriate response frame is created and sent if the RFCOMM frame contains a command message `parseControlPacket(byte[], byte, int, byte)` method is called.

All the frame types are supported and there are methods for creating each of them in `RFCOMMConnection` class. These methods are: `send_UA(byte)`, `send_DM(byte)`, `send_SABM(byte)`, `send_TEST(byte, byte[], byte)`, `send_PN(byte, byte, byte, int, int)`, `send_MSC(byte, byte, byte)` and `send_UIH(byte, byte[], int)`.

`RFCOMMConnection` class also implements two subclasses that extend the `Thread` class. These are: `SendThread` and `RecvThread` classes. These threads are using endless while loops to check the output buffers of each created DLC object (`SendThread`) and receive L2CAP packages and call the `receive_RFCOMM_Frame(byte[])` method (`RecvThread`).

`RC` and `RS` classes extend `RFCOMMConnection` class and give methods for creating L2CAP connection (socket connection at the moment), creating new DLCs and sending/reading data from input/output streams. Both classes offer the same methods.

They also starts the `SendThread` and `RecvThread` after the initial connection has been made (L2CAP link and RFCOMM control channel are opened).

Method `connect()` is used to open a new connection, meaning that the L2CAP channel is created and RFCOMM control channel has been opened (DLCI 0).

Method `openNewDLC()` is used to create a new DLC. Object of the type `DLC` is returned to the application and it is used to read/write data to the other entity.

Sequence diagram is shown on Figure 5.2.6

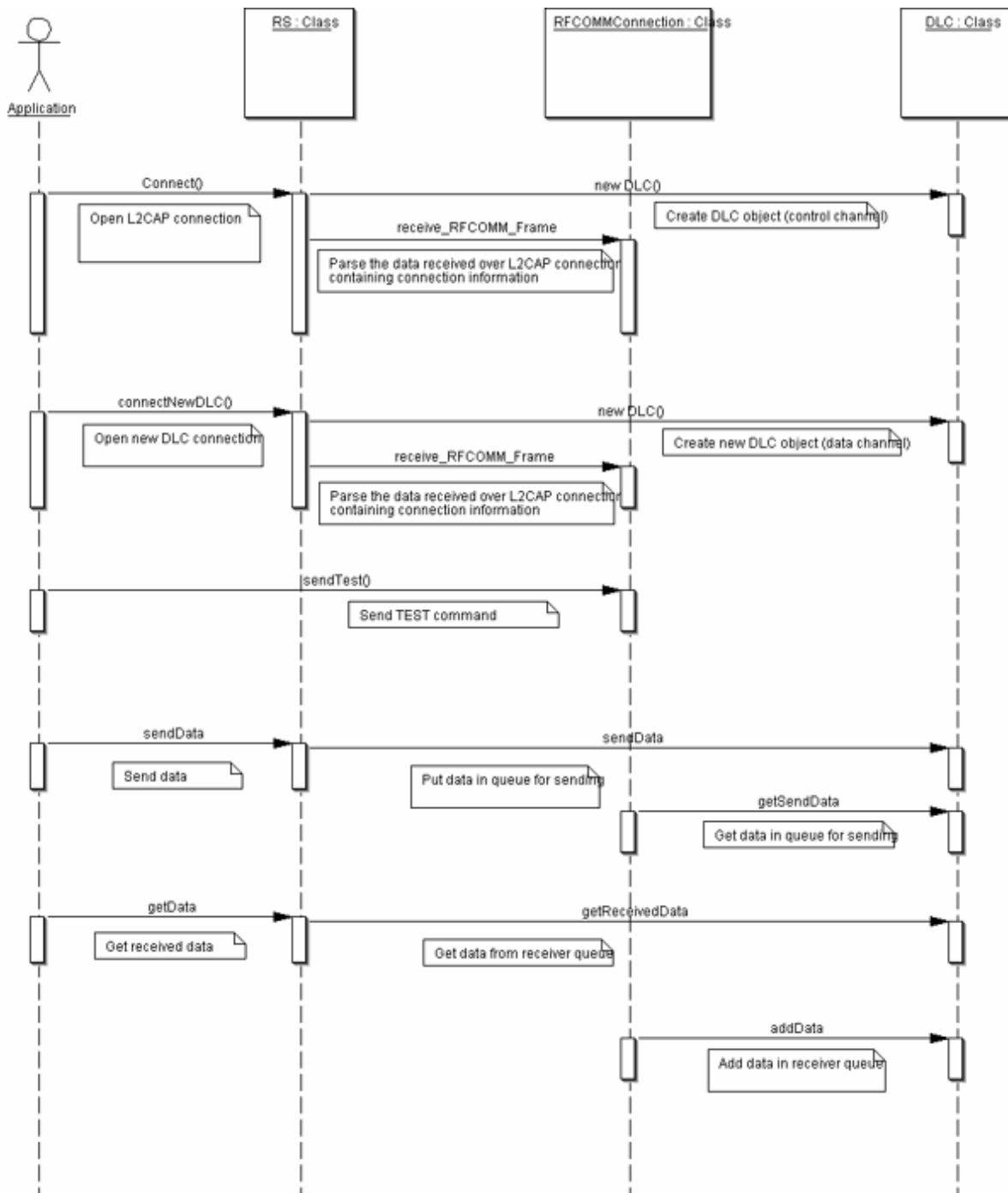


Fig. 5.2.6: RFCOMM sequence diagram

First the application should create a server/client (RS/RC class) object. By invoking the `Connect()` method of the server/client object the L2CAP connection is opened between two devices, and RFCOMM control channel is opened on that connection (new DLC object is created).

In order to send data to other entity a new DLC should be opened. This is done by invoking the `connectNewDLC()` method of the RC/RS class. This method sends and receives control packets used to establish connection and the RFCOMMConnection object. It also creates a new DLC object with given parameters. `connectNewDLC()` method returns the DLC object back to the application and this object is used for further communication with the other entity.

`send_TEST()` method of the RFCOMMConnection is used to send test command and check

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

if the DLC connection is working properly. After the test command has been sent, the same data that was sent should be received.

RS/RC objects offer two more methods. These are: `sendData()` and `getData()`:

- `sendData` calls a DLC object method `sendData()` which puts the given data in the send queue.
- `getData` calls a DLC object method `getReceivedData()` which reads the data in the receiver queue.

The parsing of received and creation of those to send L2CAP packets is done in two threads started on both sides. These threads use `addData()` and `getSendData()` methods of the DLC object. `getSendData()` is checking if there is any data in the output queue and if there is data to send, it sends it. `addData()` method is used to put the received data in the receiver queue.

5.3 javax.obex package

Package `javax.obex` is written by using JSR-82 specification. It includes interfaces and classes for OBEX communication.

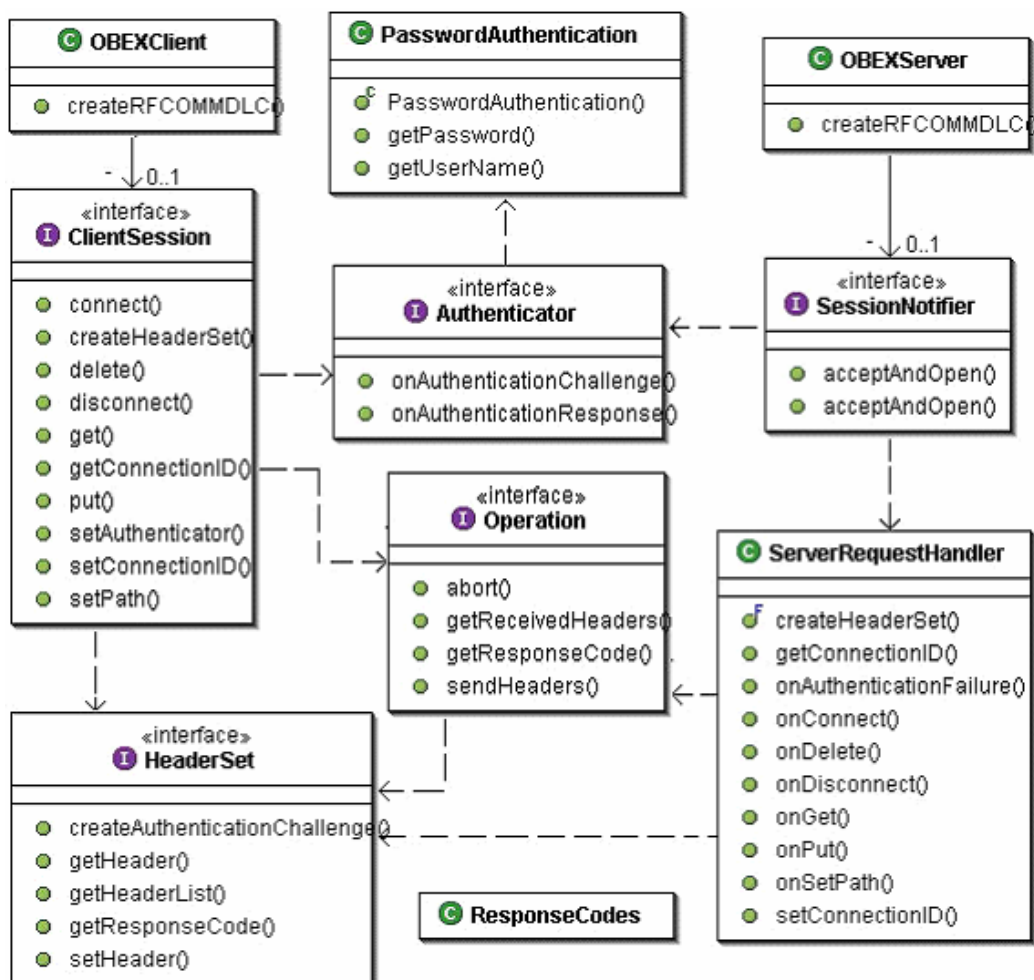


Fig. 5.3.1: javax.obex package class diagram

`ClientSession` interface provides methods for OBEX requests, and provides ways to define OBEX headers for OBEX operations CONNECT, SETPATH, PUT, GET and DISCONNECT. It extends the `javax.microedition.io.Connection` class, because it represents the OBEX connection from

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

the client's point of view.

Instance of `ClientSession` can be obtained by calling the `open(String)` method of the `javax.microedition.io.Connector` class:

```
ClientSession cs = (ClientSession)Connector.open(connectURL);
```

`HeaderSet` interface is used for define all OBEX headers (even custom headers). `setHeader(int, Object)` and `getHeader(int)` (or `getHeaderList()`) can be used for setting and getting the headers, respectively.

Instance of the `HeaderSet` can be obtained by the `createHeaderSet()` method from the `ClientSession` object:

```
HeaderSet hdr = clientSession.createHeaderSet();
```

Available header types in Bluetooth version of OBEX protocol are:

Header Value:	ID:	Data format:	Description:
COUNT	0xc0	4 byte unsigned integer	- number of objects being sent
DESCRIPTION	0x05	Unicode string	- description of the sending object
HTTP	0x47	byte sequence	- allows HTTP headers
LENGTH	0xc3	4 byte unsigned integer	- length of object in bytes
NAME	0x01	Unicode string	- name of object
OBEX_CLASS	0x4f	byte sequence	- object's class
TARGET	0x46	byte sequence	- name of the targeted service
TIME_4_BYTE	0xc4	4 byte unsigned integer	- no. of seconds since 1970-01-01
TIME_ISO_8601	0x44	byte sequence	- time header by ISO 8601 standard
TYPE	0x42	byte sequence	- type of the object
WHO	0x4a	byte sequence	- identifies the OBEX application

There can be also some custom headers, but they must follow the header specification – upper 2 bits of header ID are used to determine the type of the object:

00	Unicode string, length prefixed with 2 bytes
01	byte sequence, length prefixed with 2 bytes
10	1 byte
11	4 byte sequence – 4 byte unsigned integer

`Operation` interface provides ways to manipulate a single OBEX PUT and OBEX GET requests. After the `clientSession.put()` or `clientSession.get()` method is called, the `Operation` object is returned, so the *put* and *get* operations can be completed (after this methods are called, *Operation* object takes care for real sending or receiving the objects – completes the put or get operation).

`ResponseCodes` is class that contains all the valid response codes that an OBEX server can send to its clients.

Available response codes are: `OBEX_DATABASE_FULL`, `OBEX_DATABASE_LOCKED`, `OBEX_HTTP_ACCEPTED`, `OBEX_HTTP_BAD_GATEWAY`, `OBEX_HTTP_BAD_METHOD`, `OBEX_HTTP_BAD_REQUEST`, `OBEX_HTTP_CONFLICT`, `OBEX_HTTP_CREATED`, `OBEX_HTTP_ENTITY_TOO_LARGE`, `OBEX_HTTP_FORBIDDEN`, `OBEX_HTTP_GATEWAY_TIMEOUT`, `OBEX_HTTP_GONE`, `OBEX_HTTP_INTERNAL_ERROR`, `OBEX_HTTP_LENGTH_REQUIRED`, `OBEX_HTTP_MOVED_PERM`, `OBEX_HTTP_MOVED_TEMP`, `OBEX_HTTP_MULT_CHOICE`, `OBEX_HTTP_NO_CONTENT`, `OBEX_HTTP_NOT_ACCEPTABLE`, `OBEX_HTTP_NON_AUTHORITATIVE`, `OBEX_HTTP_NOT_FOUND`, `OBEX_HTTP_NOT_IMPLEMENTED`, `OBEX_HTTP_NOT_MODIFIED`, `OBEX_HTTP_OK`, `OBEX_HTTP_PARTIAL`, `OBEX_HTTP_PAYMENT_REQUIRED`, `OBEX_HTTP_PRECON_FAILED`, `OBEX_HTTP_PROXY_AUTH`, `OBEX_HTTP_REQ_TOO_LARGE`, `OBEX_HTTP_RESET`, `OBEX_HTTP_SEE_OTHER`, `OBEX_HTTP_TIMEOUT`, `OBEX_HTTP_UNAUTHORIZED`, `OBEX_HTTP_UNAVAILABLE`, `OBEX_HTTP_UNSUPPORTED_TYPE`, `OBEX_HTTP_USE_PROXY` and `OBEX_HTTP_VERSION`.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

`ServerRequestHandler` is class that needs to be extended before usage. It contains generic methods for defining actions when specific request arrives. The methods are: `onConnect()`, `onSetPath()`, `onPut()`, `onGet()` and `onDelete()`.

If the methods are not implemented in the class that extends `ServerRequestHandler`, then it will return predefined return values (`OBEX_HTTP_OK` for `OBEX_HTTP_NOT_IMPLEMENTED`).

`SessionNotifier` interface notifies the requests received from client, and device that wants to be a server must implement this interface and call `acceptAndOpen()` method, for passing requests to `ServerRequestHandler` (or class that extends it).

5.4 hr.fer.rasip.obex package

`hr.fer.rasip.obex` package contains implemented methods of the `javax.obex` package.

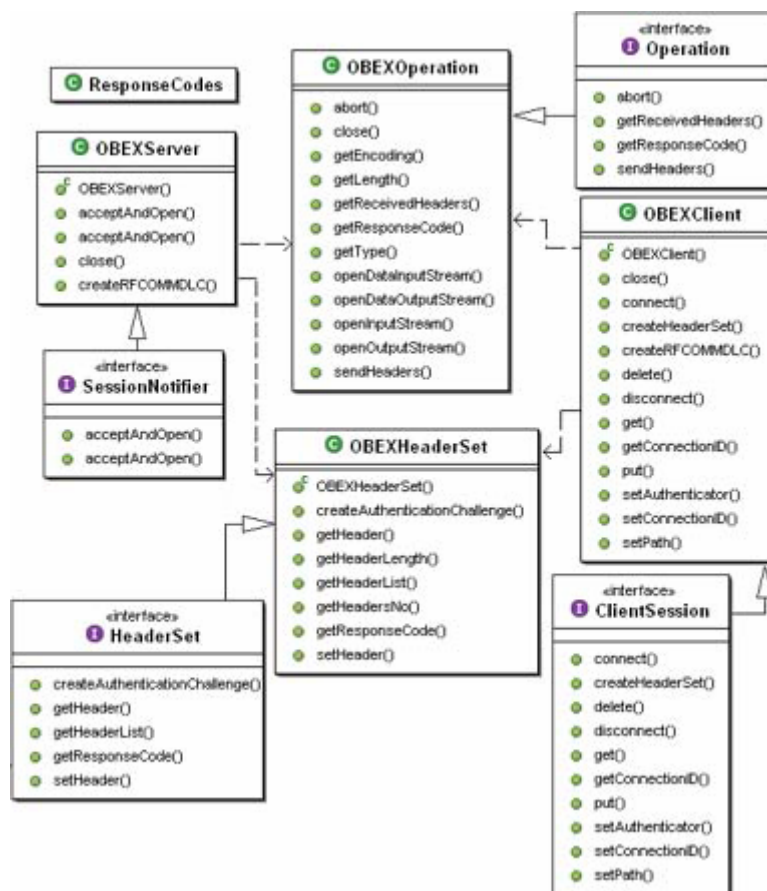


Fig. 5.4.1: `hr.fer.rasip.obex` package class diagram

At the beginning of the communication link setup, server must be initialized and wait for the client in the `SessionNotifier.acceptAndOpen(ServerRequestHandler)` method. Client also needs to be initialized.

For request and response handling on the server side, the responsible class is `javax.obex.ServerRequestHandler`. This class must be extended in order to get some functionality.

`javax.obex.SessionInterface` used for retrieving requests from the client is implemented in this package as `OBEXServer` class.

Connection starts with a `CONNECT` request that client sends to the server. For that purpose the method `ClientSession.connect(HeaderSet)` is invoked. `ClientSession` is an interface

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

defined in `javax.obex` package, and in `hr.fer.rasip.obex` is implemented as `OBEXClient` class.

Headers are stored in the `javax.obex.HeaderSet` object (implemented as `hr.fer.rasip.obex.OBEXHeaderSet` class). For sending and receiving headers, there are static methods `sendHeaders(HeaderSet, DataOutputStream)` and `receiveHeader(HeaderSet, DataInputStream)` defined in `hr.fer.rasip.obex.OBEXHeaderSet` class.

For PUT and GET operations, it is needed to instantiate the `javax.obex.Operation` object. This object is later used for completing the OBEX PUT or OBEX GET operation (sending/receiving packets with object data). `javax.obex.Operation` interface is implemented as `hr.fer.rasip.OBEXOperation` class.

Sending and receiving headers is in case of PUT and GET operations done by methods `sendHeaders(HeaderSet)` and `getReceivedHeaders()` from the `OBEXOperation` object.

DISCONNECT operation is done by issuing OBEX DISCONNECT request with `ClientSession.disconnect(HeaderSet)` method.

Upon receiving requests, `ServerRequestHandler` calls the appropriate methods: `onConnect(HeaderSet, HeaderSet)`, `onPut(Operation)`, `onGet(Operation)`, etc.

Client can get the returned response code from the server by invoking `HeaderSet.getResponseCode()` method.

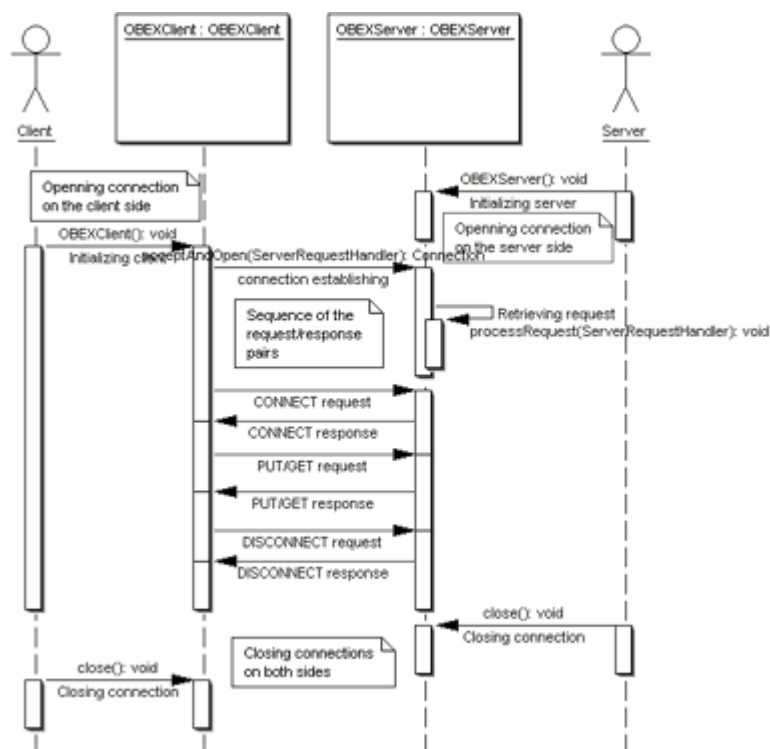


Fig. 5.4.2: `hr.fer.rasip.obex` package sequence diagram

6. JSR-82 overview

Java Specification Request 82 (JSR-82) defines the optional package for Bluetooth wireless technology for Java 2 Platform, Micro Edition (J2ME). By JSR-82 specification, Bluetooth architecture and associated APIs are defined to provide third party Bluetooth applications development.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

The API operates on top of the Connected Limited Device Configuration (CLDC) profile.

JSR-82 expert group is group that defines Java APIs for Bluetooth technology. Some of the members that participates in JSR-82 expert group are: IBM, Mitsubishi Electric, Motorola, Nokia, Research in motion, Rococo software, Sony Ericsson Mobile Communications, Sun Microsystems, Symbian, and others (for complete list, please refer to JSR-82 specification document (Links, [14])).

JSR-82 API is targeted mainly at devices that are limited in processing power and memory, and are primarily battery-operated.

Packages defined by this specification are javax.bluetooth and javax.obex. Package structure is shown on the Figure 6.0.1.

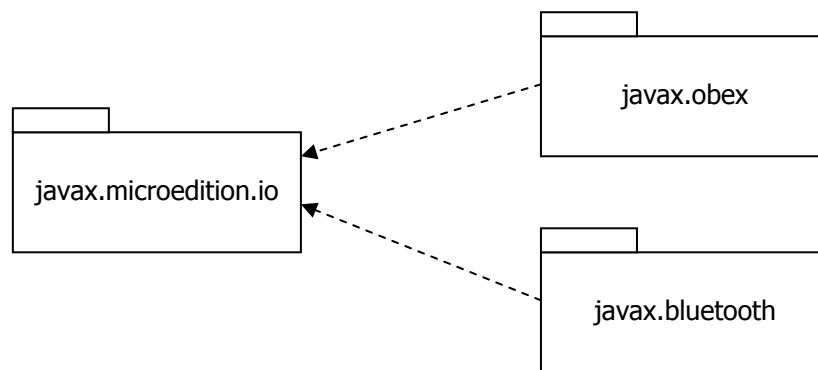


Fig. 6.0.1: JSR-82 API package structure

Mobile Information Device Profile (MIDP) devices are expected to be the first class of devices to incorporate this specification. The Bluetooth API and the MIDP APIs can coexist, but they don't depend on each other's API.

6.1 Device management

Generic Access Profile (GAP) classes contains essential Bluetooth objects such as `LocalDevice` and `RemoteDevice`. These objects provide access to and control of the local Bluetooth device, and basic information about remote device (its Bluetooth address and friendly name).

`javax.bluetooth.BluetoothStateException` class is thrown when a device cannot honor a request that it normally supports because of the radio's state.

Class `javax.bluetooth.DeviceClass` defines values for the device type and the types of services on a device.

Server can request authentication by using the `authenticate` parameter:

- if `authenticate=true`, the implementation attempts to verify the identity of every client device that attempts to connect to the service
- if `authenticate=false` or if `authenticate` parameter is not present, the implementation does not attempt to verify client devices

Server can request for encryption using the `encrypt` parameter:

- if `encrypt=true`, the implementation encrypts all communications to and from this service
- if `encrypt=false` or if `encrypt` parameter is not present, encryption is not required by

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

the server application, but it can be requested by the client application

Server can request authorization by using the `authorize` parameter:

- if `authorize=true`, the implementation consults with the BCC to determine whether or not the client device requesting a connection should be allowed access to this service
- if `authorize=false` or `authorize` parameter is not present, all clients are allowed access to this service

6.2 Discovery

Bluetooth devices can create ad-hoc networks (as stated in earlier chapters). To make this possible, they need a way to find the devices that are surrounding them. This is done by using device discovery. When a new device is found, it is added to the list of devices.

List of found devices can be obtained through an application, by invoking methods `startInquiry()` (non-blocking method) or `retrieveDevices()` (blocking method). These methods are part of the `javax.bluetooth.DiscoveryAgent`.

For using non-blocking method `startInquiry()`, discovery listener `javax.bluetooth.DiscoveryListener` must be set. When a new device is found during the inquiry, method `deviceDiscovered()` is called, and when the inquiry is completed or canceled, the `inquiryCompleted()` method will be called. To see whether the inquiry is completed or canceled, `inquiryCompleted()` method receives an argument `INQUIRY_COMPLETED`, `INQUIRY_ERROR` or `INQUIRY_TERMINATED`. Inquiry can also be canceled by calling the `cancelInquiry()` method.

When the device is found, it would be a good idea to see what can be done with it. For that purpose is used Service Discovery. Methods for service discovery are also part of `javax.bluetooth.DiscoveryAgent` package.

Client can discover services on the remote device by the Service Discovery Application Profile (SDAP). Service Discovery Protocol (SDP) combined with General Access Profile (GAP) provides the SDAP functionality.

In order to start searching for services, `DiscoveryAgent` must be instantiated:

```
DiscoveryAgent da = LocalDevice.getLocalDevice().getDiscoveryAgent();
```

Services on remote devices are identified using Universal Unique Identifier (UUID). Class `javax.bluetooth.UUID` is used to represent the available services.

Class `javax.bluetooth.DataElement` contains the various data types that a Bluetooth service attribute value can take on:

- signed or unsigned integers in the length of 1, 2, 4, 8 or 16 bytes
- String
- boolean
- UUID
- sequences of any of these scalar types

Interface `javax.bluetooth.ServiceRecord` describes a Bluetooth service to clients. It defines the Bluetooth Service Record, which contains attribute *ID*, *value* pairs. A Bluetooth attribute ID is a 16-bit unsigned integer and an attribute value is a `DataElement`.

Class `javax.bluetooth.LocalDevice` provides a `getRecord()` method that a server application can use to obtain its `ServiceRecord`. Records can be updated by calling the

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

`updateRecord()` method.

Class `javax.bluetooth.ServiceRegistrationException` represents an exception that is thrown when an attempt to modify or add a service in SDDB fails. Errors can occur:

- during the execution of `Connector.open()`;
- when a run-before-connect service invokes the `acceptAndOpen()` method and the implementation attempts to add the service record associated with the notifier to the SDDB
- after the initial creation of the service record, when the server application attempts to modify the service record in the SDDB using the `updateRecord()` method

6.3 Communication

Two Bluetooth devices that are communicating must use the same specified protocols for communication. JSR-82 specification provides API for using RFCOMM, L2CAP and OBEX protocols.

The base connection for all implemented protocol provides Generic Connection Framework (GCF) specified in the CLDC. CLDC provides following modes for opening the connection:

```
Connection Connector.open(String name);
Connection Connector.open(String name, int mode);
Connection Connector.open(String name, int mode, boolean timeout);
```

The `mode` and `timeout` parameters are optional. `String name` represents the URL of service being used (L2CAP, SPP, OBEX, etc.), `mode` parameter specifies the usage of the Connection (READ_WRITE is set by default). The last parameter `timeout`, specifies whether the timeout should be used or not.

Serial Port Profile (SPP)

Serial Port Profile specifies the formatting of the service URL:

```
btspp://host;parameters
```

```
host (client) = address:channel
```

- `address` – BT server address in length of 48 bits written using hexadecimal digits
- `channel` – channel used for communication

```
host (server) = localhost:UUID
```

- `UUID` – Universal Unique Identifier used for service identification – it length is 1 up to 128 bits written using hexadecimal digits

```
parameters (client) – possible parameters: master, encrypt and authenticate
```

```
parameters (server) – possible parameters: name, master, encrypt, authorize and authenticate
```

Parameters are delimited by ";" character. Usage for `master`, `encrypt`, `authorize` and `authenticate` – `type=bool`, where `type` is dependent on parameter (`master`, `encrypt`, `authorize`, ...). Usage for `name` – string that begins with character, and contains characters digits, spaces, dashes and underscores.

For connection establishment on the server's side, SPP creates an object of type `StreamConnectionNotifier` (example):

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

```
StreamConnectionNotifier service =
    (StreamConnectionNotifier)Connector.open(
        "btspp://localhost:123456789ABCDEF0123456789ABCDEF0;name=SPPex");
StreamConnection con = (StreamConnection) service.acceptAndOpen();
```

For connection establishment on the client's side, SPP client establishes a connection to the SPP service, but it must discover it using service discovery prior to connection establishing (example):

```
StreamConnection con =
    (StreamConnection)Connector.open("btspp://00112233445566:5");
```

"00112233445566" is the server's Bluetooth address, and the "5" is the channel identification of the SPP service. When invoking `Connector.open()` method on SPP connection, it automatically adds some service attributes to the `ServiceRecord` after creating it.

Logical Link Control and Adaptation Protocol (L2CAP)

When using L2CAP connections, `L2CAPConnectionNotifier` notifies an L2CAP server when a client initiates a connection. Once the connection is created, `L2CAPConnection` object is returned.

Connection-oriented channels needs to be configured once the connection is established. Parameters that are configured are: Maximum Transmission Unit (MTU) – the default value is 672 bytes, Flush timeout (default value is 0xFFFF), and Quality of Service (QoS).

L2CAP specifies the formatting of the service URL:

```
btl2cap://host;parameters
```

host (client) = address:psm

- address – BT server address in length of 48 bits written using hexadecimal digits
- psm – protocol/service multiplexer

host (server) = localhost:UUID

- UUID – Universal Unique Identifier used for service identification – it length is 1 up to 128 bits written using hexadecimal digits

parameters (client) – possible parameters: master, encrypt, authenticate, receiveMTU and transmitMTU

parameters (server) – possible parameters: name, master, encrypt, authorize, authenticate, receiveMTU and transmitMTU

Parameters are delimited by semicolons. Usage for master, encrypt, authorize and authenticate – type=bool, where type is dependent on parameter (master, encrypt, authorize, ...). Usage for name – name=string, where string begins with a character, and contains characters digits, spaces, dashes and underscores. Usage for receiveMTU and transmitMTU – decimal digits.

Connection establishment on the client's side (example):

```
L2CAPConnection client = (L2CAPConnection) Connector.open(
    "btl2cap://0123456789012;ReceiveMTU=512;TransmitMTU=512");
```

Connection establishment on the server's side (example):

```
L2CAPConnectionNotifier server = (L2CAPConnectionNotifier)
```

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

```
Connector.open("bt12cap://localhost:123456789ABCDEF0123456789ABCDEF0;
name=L2CAPex");

L2CAPConnection con = (L2CAPConnection)server.acceptAndOpen();
```

After successful connection, ServiceRecord is created.

To obtain maximum MTU supported by the stack application must use method `LocalDevice.getProperty("bluetooth.l2cap.receiveMTU.max")`;

If L2CAP connection fails, application must throw a `BluetoothStateException`.

Object EXchange protocol

Notifier method for OBEX protocol is called `SessionNotifier`. When the client requests for communication, `SessionNotifier.acceptAndOpen()` method is invoked, and the communication can be established. If there is an error while calling `Connector.open()` method, the `ConnectionNotFoundException` exception is thrown.

OBEX over RFCOMM is implemented as the Generic Object Exchange Profile (GOEP), by the Bluetooth specification.

Address formatting used here is `btgoep://host;parameter`. Host and parameter are constructed in the same way as for the SPP.

Connection establishment on the client's side (example):

```
ClientSession
con=(ClientSession)Connector.open("btgoep://123456789012:5");
```

Connection establishment on the server's side (example):

```
ServerRequestHandler server = new ServerRequestHandler();
SessionNotifier sn=(SessionNotifier)Connector.open(
"btgoep://localhost:123456789ABCDEF0123456789ABCDEF0");
sn.acceptAndOpen(server);
```

OBEX over TCP/IP – protocol used here is `tcpobex`, so the address formatting is like `tcpobex://host;parameter`. Host, in this case, is IP address of the target device, optionally followed by the ":" character, and the port. Default port is 650.

Example of the OBEX TCP/IP address: `"tcpobex//123.45.67.89:1234"`

7. Installation guide – software

This project is a library intended to be integrated into Java applications that use Bluetooth, and (for now) it's distributed in the form of Eclipse project. The [Eclipse Platform](#) is an open extensible IDE for anything, but the most evolved part of the platform are Java Development Tools (JDT). Eclipse platform along with JDT offers currently the most usable general-purpose Java IDE.

Common procedures related to Eclipse platform (importing projects, creating run configurations, configuring libraries) are described later on in this document, in order to help the novice users through their first steps in the Eclipse.

Very important thing about Java Bluetooth Stack is hardware. JBS is a hardware dependent project, and very little can be done without it. Currently JBS supports only H4 (UART) HCI Transport Layer (for some supported devices see Hardware installation guide in the next Chapter) through the Java commAPI, so, in order to get the test configuration going, Java commAPI needs to be installed correctly (pay special attention to this, as this can be the source of many problems afterwards).

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Besides that, some kind of environment supporting MIDP profile must be installed. Since JSR-82 specification is targeted at MIDP devices, JBS needs some J2ME classes to function completely (this goes mostly for javax.microedition.rms package, which is used only to store information about trusted devices). Sun's Wireless Toolkit can be installed for this purpose (after installation just put `midpapi.zip` in project references) or just install ME4SE (MicroEdition4StandardEdition) library (put `me4se.jar` and `png.jar` in project references). ME4SE is a library that enables a user to run a midlet (applet written for J2ME) on a J2SE platform (see [link](#) for details).

7.1 Installing Eclipse Platform and Java commAPI and configuring environment variables for commAPI

7.1.1 Eclipse Platform installation

For Eclipse install instructions, and any issues regarding Eclipse platform, please refer to the Eclipse project FAQ (<http://www.eclipse.org/eclipse/faq/eclipse-faq.html>).

7.1.2 Sun Wireless ToolKit installation

Installing Sun WTK (Wireless ToolKit) is pretty straightforward, just download and run the executable installation file. Follow the on-screen instructions, and note the directory in which the WTK will be installed, because it will be needed later on in configuring project references (`midpapi.zip` which needs to be in project references, is usually in the `WTK_install_dir\lib\` subdirectory).

7.1.3 Installing commAPI

Please refer to Appendix A: Getting commAPI to work

7.1.4 Importing project into workspace

Please refer to the Appendix B: Importing projects into workspace

7.1.5 Running samples and JUnit tests

Once the project is imported and all required libraries in place, it is possible to explore the samples (before that, make sure you have installed some Bluetooth hardware with H4 (UART) Transport Layer – refer to the hardware install instructions (next chapter of this documentation) for more detail).

For creating new Run Configuration please refer to the Appendix C, and for creating new JUnit Run Configuration refer to the appropriate section in Appendix E.

8. Installation guide – hardware

Currently, Java Bluetooth Stack supports only Bluetooth devices that use H4 (UART) Host transport layer to communicate with Hosts (typically PCs). Support for those devices is accomplished through Java Communication API (for instructions on installing the commAPI, please refer to the Appendix A: Getting commAPI to work).

In order to use the stack, you will need to have at least two Bluetooth devices with UART transport layer enabled. These devices are typically Bluetooth serial modules such as ROK101008 from Ericsson Application Toolkit (other companies that manufacture serial modules are Brainboxes, SMART Modular Technologies, etc.).

Some PC-Cards (PCMCIA) might work, as long as they are recognized by the OS as COM ports. This heavily



Fig. 8.1: Xircom CBT PC-Card installed as a virtual COM port

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

depends on the OS you're using, i.e. Xircom CBT PC-Card is installed by Windows 2000 as virtual COM port, while Windows XP configure it differently, rendering it unusable with JBS.

Important notice: following instructions apply only to Bluetooth hardware noted here, which has been tested exhaustively. They might work for other similar hardware, but we cannot take ANY responsibility for any damage caused by improper Bluetooth hardware installation.

8.1 Xircom CBT

General notes

You should start by downloading the driver from the [Xircom](#) site. After you download the driver, run it, follow the instructions and install it into a temporary directory (but make sure you remember which one).

Xircom CreditCard Bluetooth Adapter install instructions for notebook computers (computers with build-in PC-Card slot)

Insert the card into the PC-Card slot. Windows should detect and ask for drivers. Point them to the location where you have previously unpacked (installed) downloaded drivers, and wait until all the profiles (virtual COM ports) are installed.



Fig. 8.2: Xircom CBT PC-Card

[BUG] Even if the "Restart computer now" message box appear, please wait until you're absolutely certain that the driver installation has ended (i.e. there's no disk activity for a period of 3 minutes), and then and only then restart your computer. This is a known issue and is documented in the driver installation notes.

Xircom CreditCard Bluetooth Adapter install instructions for regular PCs


If you need to install any PC-Card to a standard PC, you need to have a PC-Card adapter. These instructions refer to the Ricoh PC-Card adapter, but the procedure should work with any similar PC-Card Adapter.

First, install the drivers from the installation diskette. Shutdown your computer and install the PC-Card adapter into a free PCI slot. Turn your computer on, Windows should recognize newly installed adapter and finish the installation.

Restart your computer, and when Windows finish booting insert Xircom CBT into Ricoh PC-Card adapter. Windows will prompt you for drivers. Follow the instructions given above (installation for notebook computers).

a. Post install notes

The last thing to do in order to use Xircom CBT with JBS, is to open Device Manager and, under Ports (COM & LPT) tree node you should see something like "Xircom CreditCard Bluetooth Adapter (COMX)" (see picture above). You should note the COM port number, as you will use it in initializing the BCC (Bluetooth Control Center) in every JBS application.

Before you run any of the samples included with the stack using COM port number assigned to Xircom CBT, you must disable the Xircom Bluetooth environment application (in the tray menu right click the  icon and uncheck "Radio On", or choose "Exit" from the pop-up menu), otherwise you will get Port in use Exception.

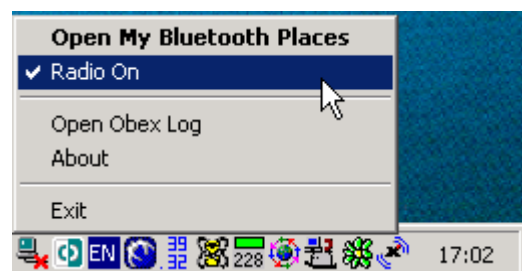


Fig. 8.3: BT tray icon

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

8.2 Generic Bluetooth serial module install instructions

There are two main things you should provide to a serial module – power and connection to serial port. As they consume a significant amount of current (as much as 150 mA during inquiry), they need a separate power source (serial port can only provide about 25 mW of power). If serial module supports USB besides UART, you can connect it to the PC with the USB cable provided with the module. If it doesn't, you must consult the module documentation to find out what voltage level the module requires. If that's 5V DC or 12 V DC, you can use standard PC AT power supply.

[Caution: messing around with 220 V AC can be very dangerous if you don't know exactly what are you doing!]

a. Ericsson ROK 101008 install instructions

In the package with the module you should have received crossover serial cable (female DB9 <-> female DB9 connectors), serial cable adapter (for board header connector) and a USB cable. Connect the serial cable to a free COM port, on the other end connect the serial cable adapter, and connect the adapter to the module board. Connect the module with the USB cable to the PC to provide power.

Module is now ready for use with JBS; specify the COM port number in the BCC (Bluetooth Control Center) initialization code.

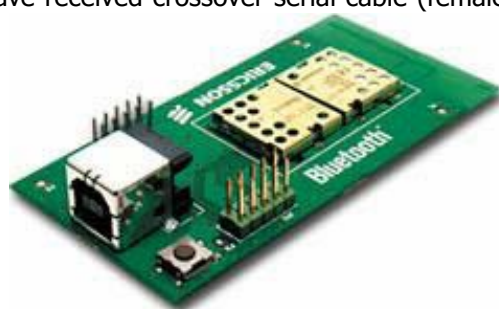


Fig. 8.4: Ericsson ROK 101008 BT module

b. ROK104001 module installation instructions

In the package with the module you should have received straight serial cable (female DB9 <-> male DB9 connectors). Connect the serial cable to a free COM port and to the module board. Connect the module to a standard PC power supply (either from a separate power supply, or from your computers'), or to any other power source that provides 6 – 20 V DC using the DC plug connector.

Module is now ready for use with JBS; specify the COM port number in the BCC (Bluetooth Control Center) initialization code.

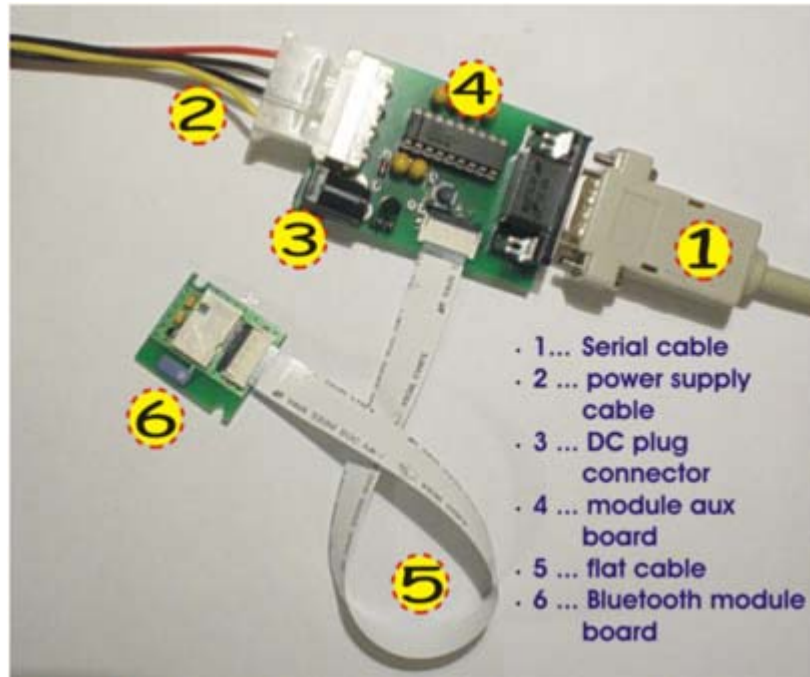


Fig. 8.5: ROK 104001 Module installation

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Appendix A: Getting commAPI to work

- download javacomm_20-win32.zip from [commAPI](#) home page
Unzip the file javacomm20-win32.zip. This will produce a hierarchy with a top-level directory commAPI

The examples in this document assume that you have unzipped the javacomm20-win32.zip file in your C: partition and your JDK installation is in C:\j2sdk1.4.2_02.

If you have installed JDK in another location or unzipped javacomm20-win32.zip in another location modify the example commands appropriately.

Copy win32com.dll to your <JDK>\bin and to your <JDK>\jre\bin directory

```
C:\>copy c:\commapi\win32com.dll to c:\j2sdk1.4.2_02\bin
```

```
C:\>copy c:\commapi\win32com.dll to c:\j2sdk1.4.2_02\jre\bin
```

Copy comm.jar to your <JDK>\lib and to your <JDK>\jre\lib directory

```
C:\>copy c:\commapi\comm.jar c:\j2sdk1.4.2_02\lib
```

```
C:\>copy c:\commapi\comm.jar c:\j2sdk1.4.2_02\jre\lib
```

Copy javax.comm.properties to your <JDK>\lib and to your <JDK>\jre\lib directory

```
C:\>copy c:\commapi\javax.comm.properties c:\j2sdk1.4.2_02\lib
```

```
C:\>copy c:\commapi\javax.comm.properties c:\j2sdk1.4.2_02\jre\lib
```

The javax.comm.properties file must be installed. If it is not, no ports will be found by the system.

Set your path variables like this (this is probably the most important part):

[Beware, this jars are the ones installed by J2SDK 1.4.2_02, if you're configuring any other J2SDK version with this commapi please check jar names and change them accordingly.]

```
JAVA_HOME=<JDK>\
```

```
CLASSPATH=%JAVA_HOME%\lib\comm.jar;%JAVA_HOME%\jre\lib\comm.jar;%JAVA_HOME%\jre\lib\rt.jar;%JAVA_HOME%\jre\lib\jsse.jar;%JAVA_HOME%\jre\lib\charsets.jar;%JAVA_HOME%\jre\lib\jce.jar;%JAVA_HOME%\jre\lib\charsets.jar;%JAVA_HOME%\jre\lib\plugin.jar;%JAVA_HOME%\jre\lib\sunrsasign.jar;%JAVA_HOME%\lib\dt.jar;%JAVA_HOME%\lib\htmlconverter.jar;%JAVA_HOME%\lib\tools.jar
```

PATH=%JAVA_HOME%\bin;%PATH% (please note that "%JAVA_HOME%\bin;" must be at the very beginning of the path variable)

Restart your machine, and to test your newly installed/configured Java commAPI run Blackbox example from commAPI installation directory:

Go to the samples\BlackBox subdirectory of the directory you unpacked commapi:

```
C:\>cd commapi\samples\BlackBox
```

Start the BlackBox sample giving the path to the comm.jar as the classpath parameter:

```
C:\commapi\samples\BlackBox>java -cp .;c:\j2sdk1.4.2_02\lib\comm.jar BlackBox
```

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

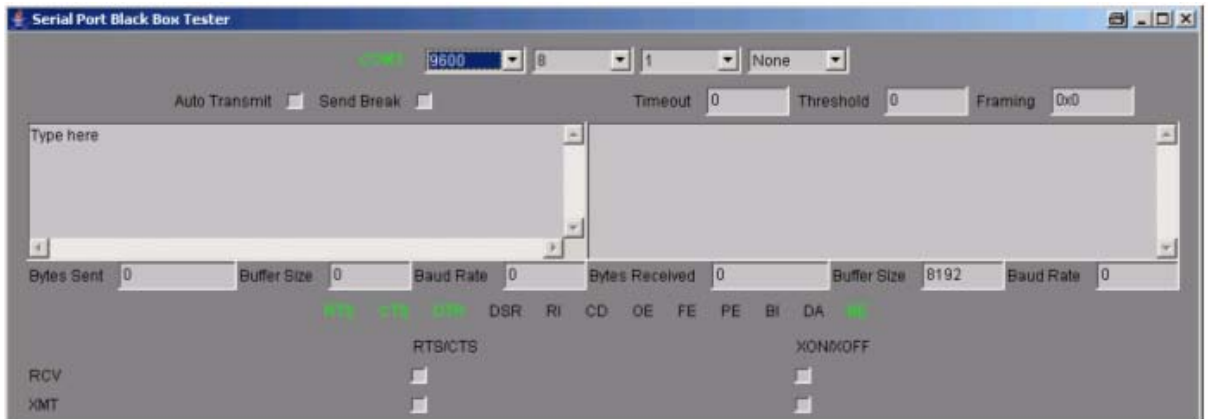


Fig. A.1: Blackbox example main screen

Figure A.1 shows the Blackbox example main screen that you should get if the java and commAPI are installed correctly.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Appendix B: Importing existing projects into workspace

After you have got commapi to work, download the project zip and unpack it into directory of your choice (just make sure you remember which one is it). Start Eclipse Platform and follow the instructions for importing existing projects into workspace and import the JBS into your workspace.

If you imported jbs project correctly, package explorer view should look like on Fig. B.1

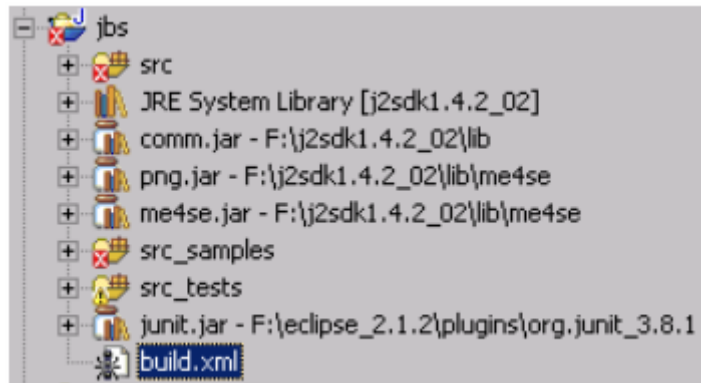


Fig. B.1: Package explorer view after importing JBS project

When you're done with importing, follow the instructions for repairing project references (instructions in Appendix C) – they have to contain `comm.jar`, `junit.jar` (found in the `eclipse_install_directory\plugins\org.junit_X.X.X`, `X.X.X` denotes version), and `midpapi.zip` (or combination of `me4se.jar` and `png.jar` instead of `midpapi.zip`).

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Appendix C: Creating Eclipse Runtime configurations

The most practical way to create Eclipse Runtime configuration is to just let the Eclipse handle all the work:

open the java file for which is intended to create runtime configuration into the editor of the Eclipse platform

choose **Run -> Run ...** from the menu bar, and in the window that is opened right click on the **Java Application** in the left pane. Select the **New** option (Fig. A.1)

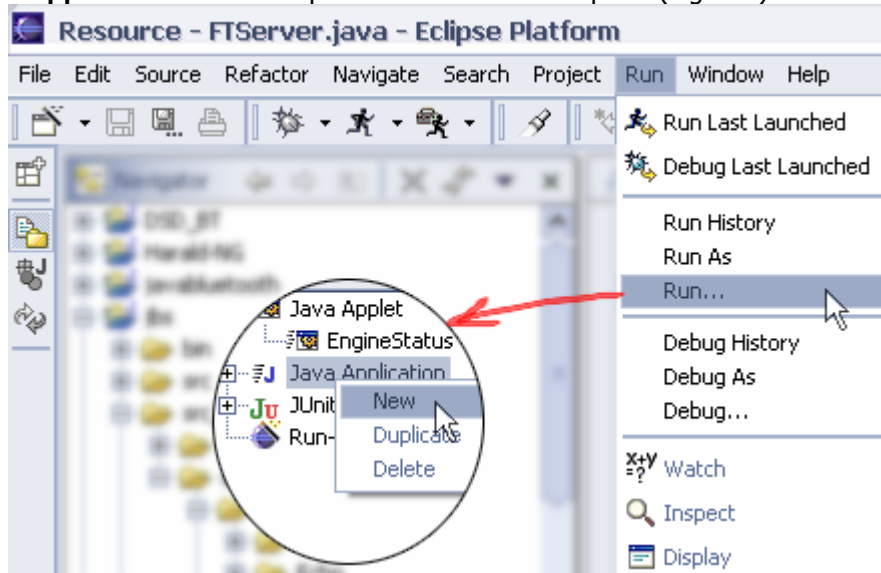


Fig. C.1: Creating the new Runtime Configuration

After that we get the window (similar to window on Fig. E.7) with parameters set to the corresponding values – those values can be changed if needed

Click on the **Run** button to run newly created run configuration

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Appendix D: Checking and configuring project references

Projects can contain more of external libraries, so it can easily happen that some error occur in the link to those libraries. The most usual error is wrong path to the external library – this can happen if the project is transferred onto another computer, where the paths to the libraries are not the same as on the computer where the libraries are linked with project. Other case in which the path error can occur is if someone accidentally deletes or uninstall some of the libraries.

No matter what the error is, we can always check (and correct those errors) by right-clicking on the projects root in the Navigator view and selecting Properties in the drop-down menu. In the opened window select the **Java Build Path** option in the left pane, and the **Libraries** on the tab in the right pane (Fig. 6.1.13.)

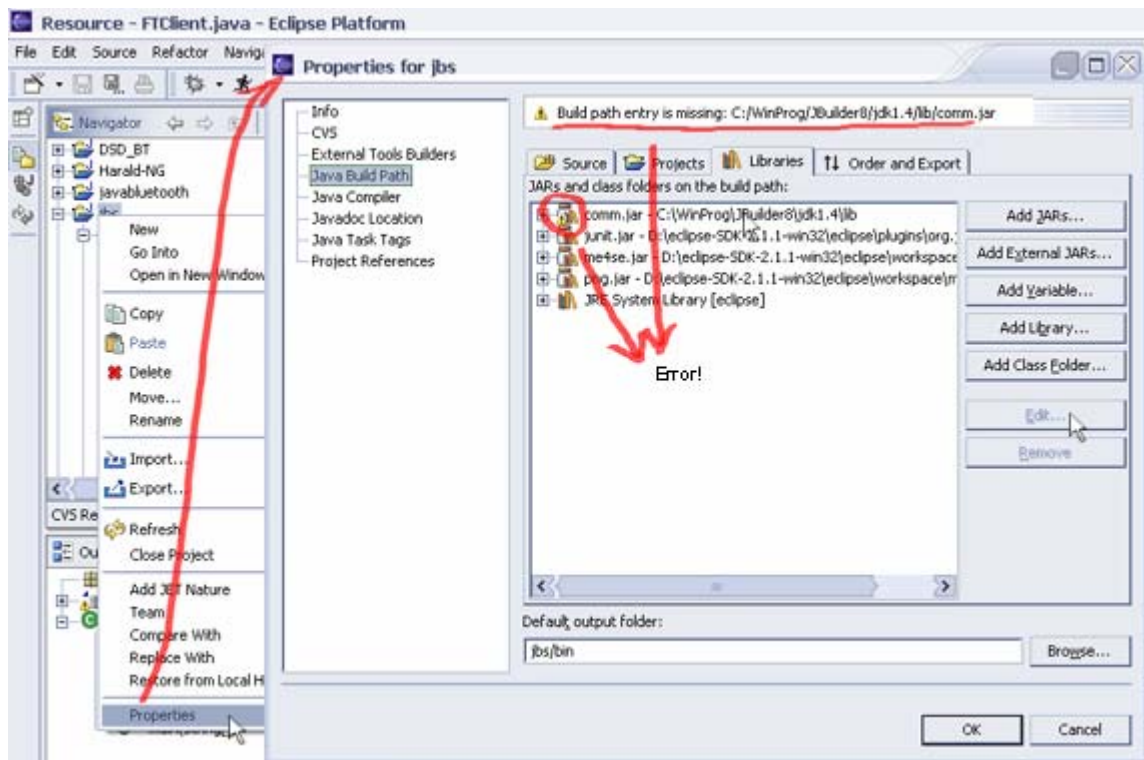


Fig. D.1: Project's properties – Java Build Path

As shown on Figure D.1. there is an error with the comm.jar library path – error message is shown in the bar above. To correct this error, select the library line path (where is the yellow exclamation sign) and click on **Edit** button on the right side of the Properties window. In here is possible to manually find the missing file.

Common locations for libraries used by Java Bluetooth Stack:

```

comm.jar      <JAVA_HOME>\lib
junit.jar    <ECLIPSE_HOME>\plugins\org.junit_X.X.X (in this case X.X.X is 3.8.1)
midpapi.zip  <WTK_HOME>\lib

```

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Appendix E: Eclipse JUnit support

Writing JUnit tests:

Eclipse installation includes JUnit in the `org.junit` plug-in. Since Eclipse version 2.0 this plug-in is part of the build.

Before any test can be written or run, `junit.jar` library must be added to the build class-path:

right-click on the project's root directory in the Navigator pane and select **Properties**

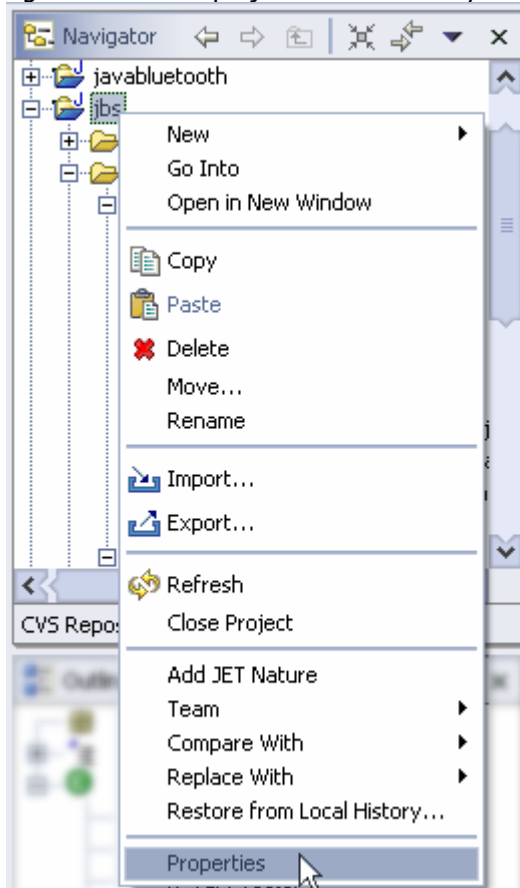


Fig. E.1: Opening project's properties

In the opened window select **Java Build Path** in the left pane, and the **Libraries** tab-pane on the right side of the window. New external libraries are added by **Add External JARs** button (Fig. E.2)

From the JAR selection window (Fig. E.2) in the **Look-in** path select path to the `<ECLIPSE_HOME>\plugins\org.junit_3.8.1` where is JUnit plug-in located, select the `junit.jar` file and open it

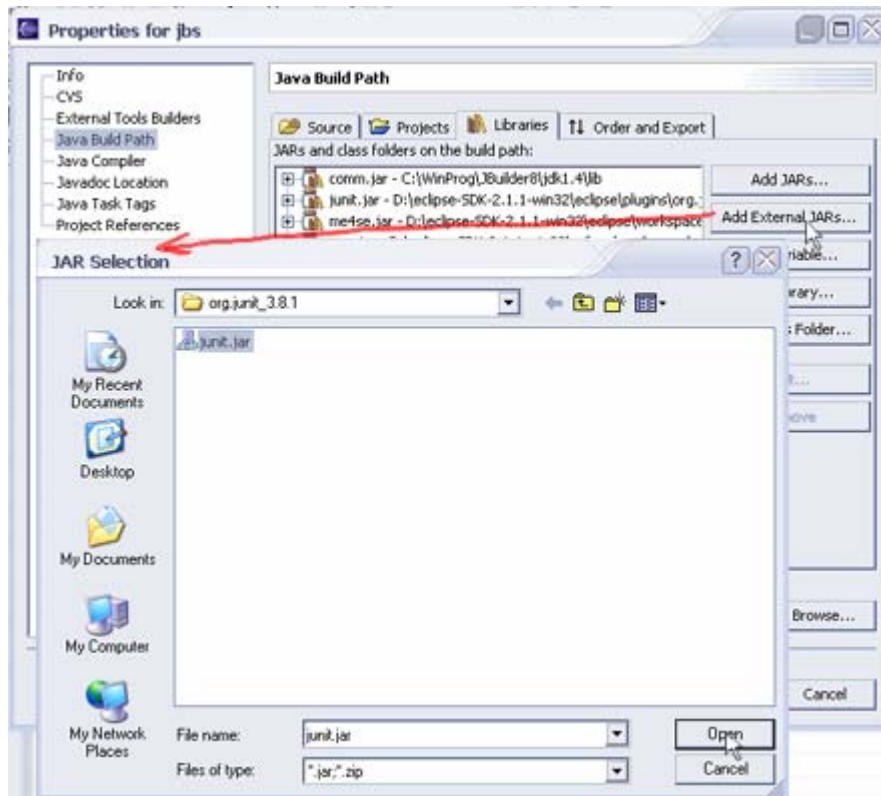


Fig. E.2: Adding *junit.jar* to the build class-path

Click on the OK button in project's properties window to confirm properties of the project
 Now when the libraries are set, it is possible to write some JUnit tests. All the JUnit tests must be the subclasses of the *TestCase* class.

The easiest way of writing JUnit tests is by using wizard:

open the New wizard (**File -> New -> Other ...**), select **Java -> JUnit** in the left pane, and the *TestCase* in the right pane, and click **Next** (Fig. E.3)

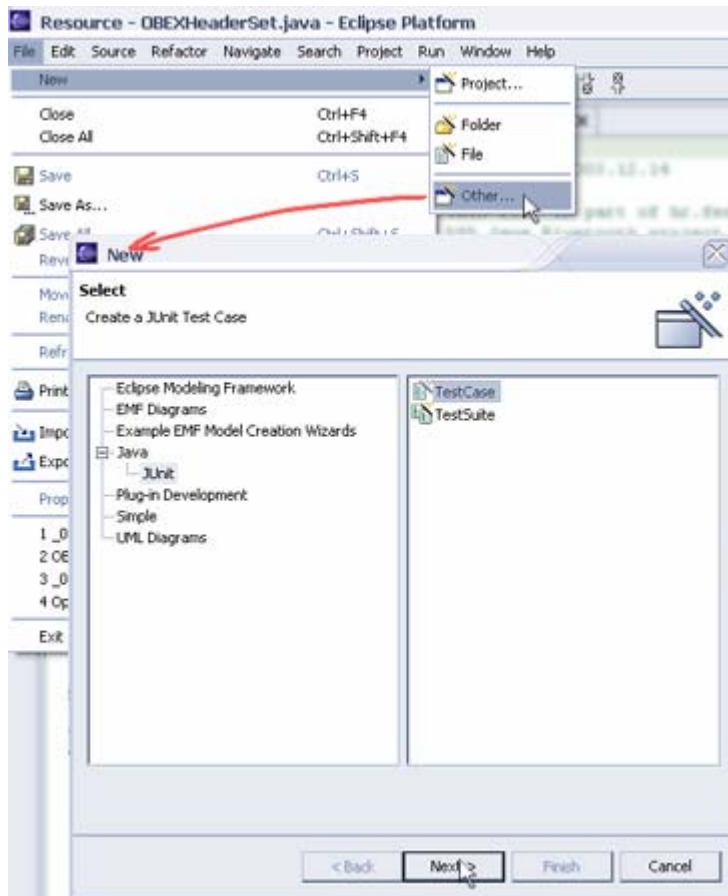


Fig. E.3: Using wizard for writing new JUnit TestCase or TestSuite

Enter *TestFailure* as the name of your test class



Fig. E.4: Creating a new JUnit TestCase

Click **Finish** to create the test class

Add the testing procedure to the test class

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Running JUnit tests

open the test in the editor view (in Eclipse)

activate the **Run** drop-down menu in the toolbar and select **Run as -> JUnit Test** (alternative is to create a new JUnit Run Configuration – see next section for details)

The JUnit test window now shows you the test run progress and status

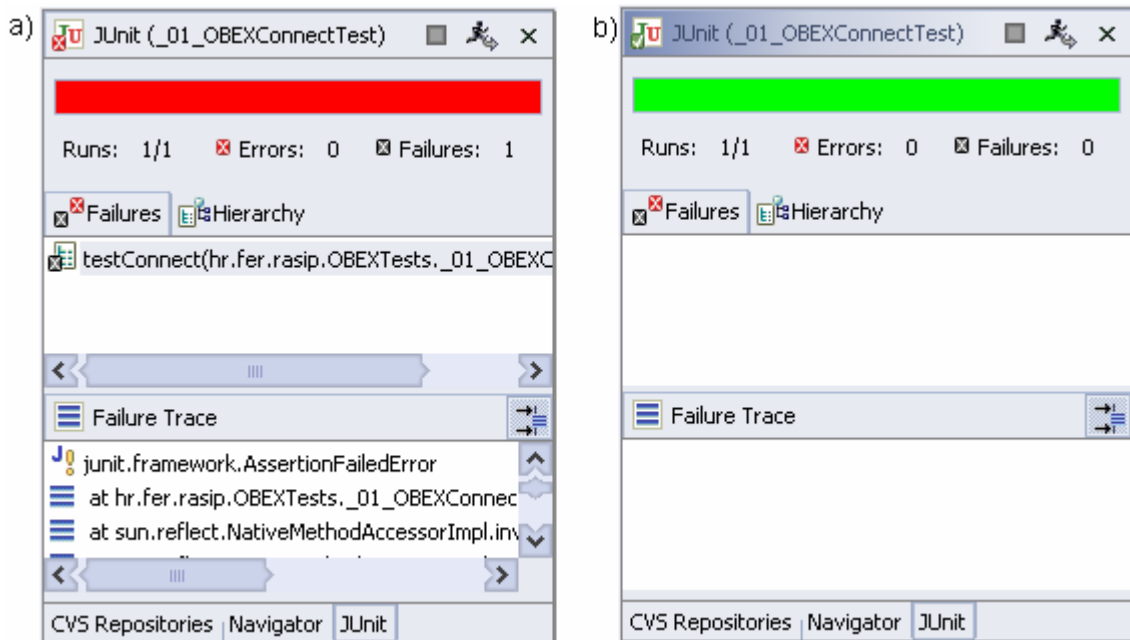





Fig. E.5: JUnit test run status: a) test failed, b) test successfully completed

Fig. E.5 shows test run status – in case a), test failed (icon  is shown in the upper-left corner of JUnit view). Description of why test failed is provided in the "Failure Trace" pane (in this case, assert test method failed) – you can click on any of the lines in this pane to go to the specific line in code to see why test failed. In case b), test has successfully finished (icon  is shown).

The JUnit view has two tabs: one shows you a list of failures and the other shows you the full test suite as a tree.

Test can be re-run either by the method previously described (Run as -> JUnit Test), by JUnit Run Configuration, or by Re-Run button () in the JUnit view.

It is possible to Run JUnit test on one or more tests at once:

Running all tests at once: select a package or source folder and run all included tests by **Run as -> JUnit test**

Run a single test method: select a test method in the Outline or Package Explorer and run test by **Run as -> JUnit test**

Re-run a single test: select a test in the JUnit view and execute **Rerun** from the context menu

Debugging the test failure:

double click the failure entry from the stack trace in the JUnit view to open the corresponding file in the editor

set the breakpoint at the beginning of the test method

select the test case and execute **Debug as -> JUnit Test** from the **Debug** drop-down menu

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Creating a new JUnit Run Configuration

The most practical way to run JUnit tests is to just let the Eclipse handle the testing: open the JUnit test file in the editor of the Eclipse platform choose **Run -> Run** from menu bar, and in the window that is opened right-click on the JUnit configuration and select **New** (Fig. E.6)

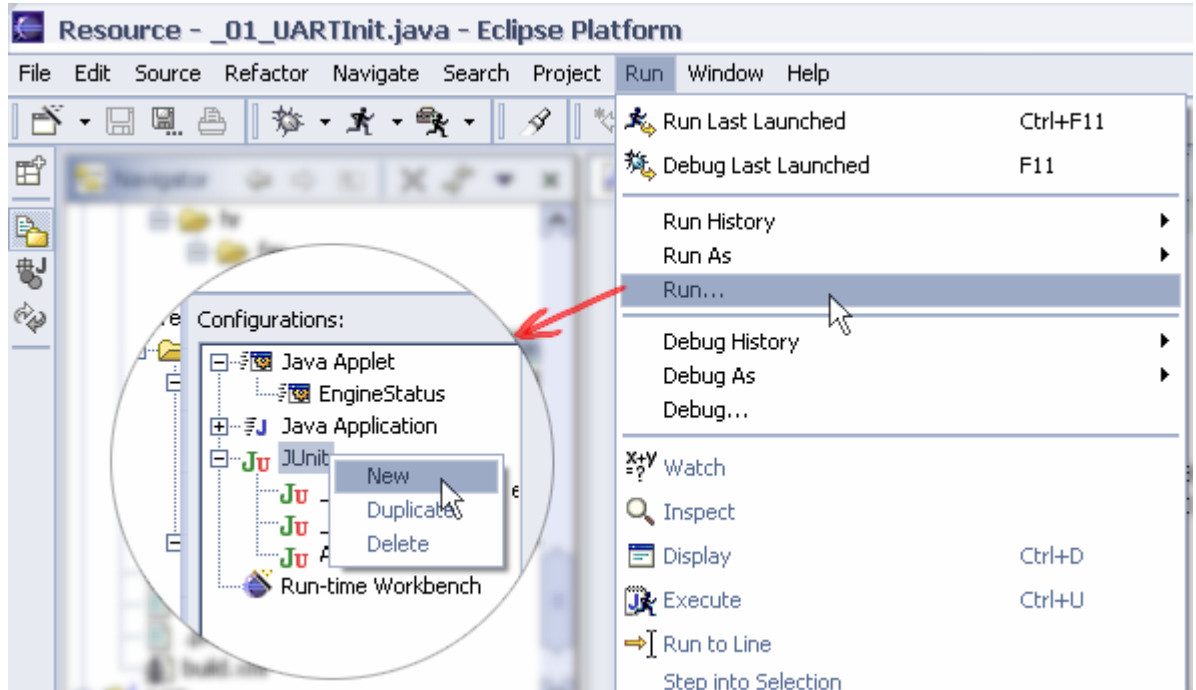


Fig. E.6: Creating new JUnit run configuration

JUnit run configuration is now automatically created. You may now click on the **Run** button to start the test, or view/change some parameters if needed (Fig. C.7)

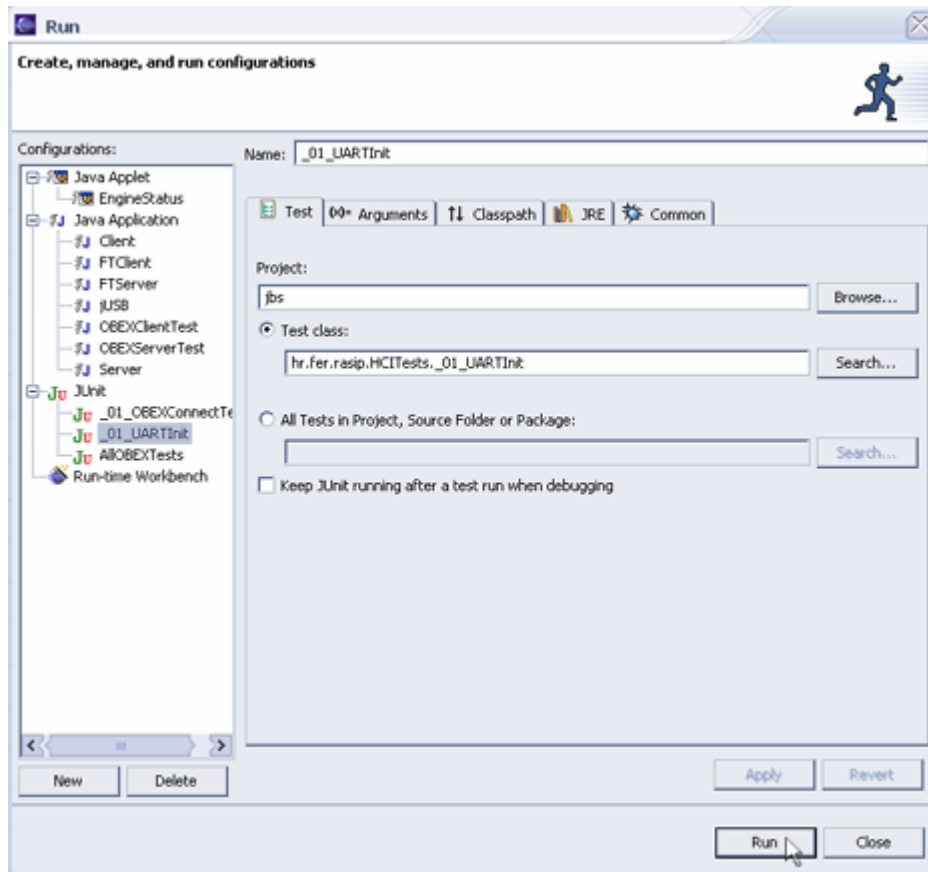


Fig. E.7: JUnit Run test created and ready to run

Creating a JUnit test suite

If someone wants to run all (or several) tests at once, there should be some automated procedure for that, because it can be pretty annoying to run tests one by one. For that purpose, JUnit has implemented the test suite.

The easiest way of creating a JUnit test suite is by using wizard:
open the New wizard (**File -> New -> Other ...**)

Select **Java -> JUnit** in the left pane and **TestSuite** in the right pane and click **Next** (Fig.

E.8)

enter a name for your test suite class (the convention is to use "AllTests" or similar)



Fig. E.8: Creating a new JUnit TestSuite

select the classes that should be included in the suite

You can add or remove test classes from the test suite:
 manually, by editing the test suite file
 by re-running the wizard and selecting the new set of test classes

NOTE: the wizard put markers `//&JUnit-BEGIN&` and `//&JUnit-END&`, into the created Test Suite class, which allows the wizard to update existing test suite classes. Editing code between these two markers is not recommended.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Appendix F: Eclipse & CVS

When your CVS server supports only SSHv2 (that's a good thing actually) and Eclipse has a built-in client only for an SSHv1 (*extssh* connection method), there are two options:

Download [CVS-SSH2 Plug-in for Eclipse](#) (an [Eclipse](#) plug-in that allows CVS access on SSH2 session)

Use *ext* connection method and an external ssh util (like *plink* from creators of *putty*) – this method involves generating RSA key pair and storing the public key on the server – it's pretty complex...

I suggest the first method – download the plug-in, kill eclipse if it's running, unzip the plug-in into the eclipse\plugins, and start Eclipse again.

Main CVS operations:

Creating a CVS repository location – every active developer in the team will have to do this eventually

Sharing a new project using CVS – this has to be done just once, by whoever has started coding or created Eclipse project – let's call him/her the initiator from now on

Checking out a project from a CVS repository – every developer in the team (except the initiator must do this to get a local copy of the project)

Updating code – done when needed

Committing code – done after some changes in code

Synchronizing with a repository – local copy is compared with repository copy

[Following chapters have been taken out of Eclipse documentation and are somewhat shortened and fulfilled with pictures]

Creating a CVS repository location

Open the CVS Repositories view by selecting **Window -> Show View -> Other...** on the main menu bar and then selecting **CVS -> CVS Repositories** from the Show View dialog. Alternatively, the CVS Repositories view is also shown in the CVS Repository Exploring perspective.

From the context menu of the CVS Repositories View, select **New -> Repository Location**. The Add CVS Repository wizard opens.

Enter the information required to identify and connect to the repository location:

In the **Host** field, type the address of the host.

In the **Repository path** field, type the path to the repository on the host (in our case something like `/var/CVSRROOT/<project_name>`)

In the **User** field, type the user name under which you want to connect to the repository.

In the **Password** field, type the password for the above user name.

From the **Connection Type** list, select the authentication protocol of the CVS server. There are three connection methods that come with the Eclipse CVS client:

pserver - a CVS specific connection method (for read only access).

extssh - an SSH 1.0 client included with Eclipse

ext - the CVS ext connection method that uses an external tool such as SSH to connect to the repository. The tool used by ext is configured in the **Team -> CVS -> EXT Connection Method** preference page.

extssh2 – a method added by CVS-SSH2 Plug-in

If the host uses a custom port, enable **Use Port** and enter the port number.

(Optional) Select **Validate Connection on Finish** if you want to authenticate the specified user to the specified host when you close this wizard. (If you do not select this option, the user name will be authenticated later, when you try to access the contents of the repository.)

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

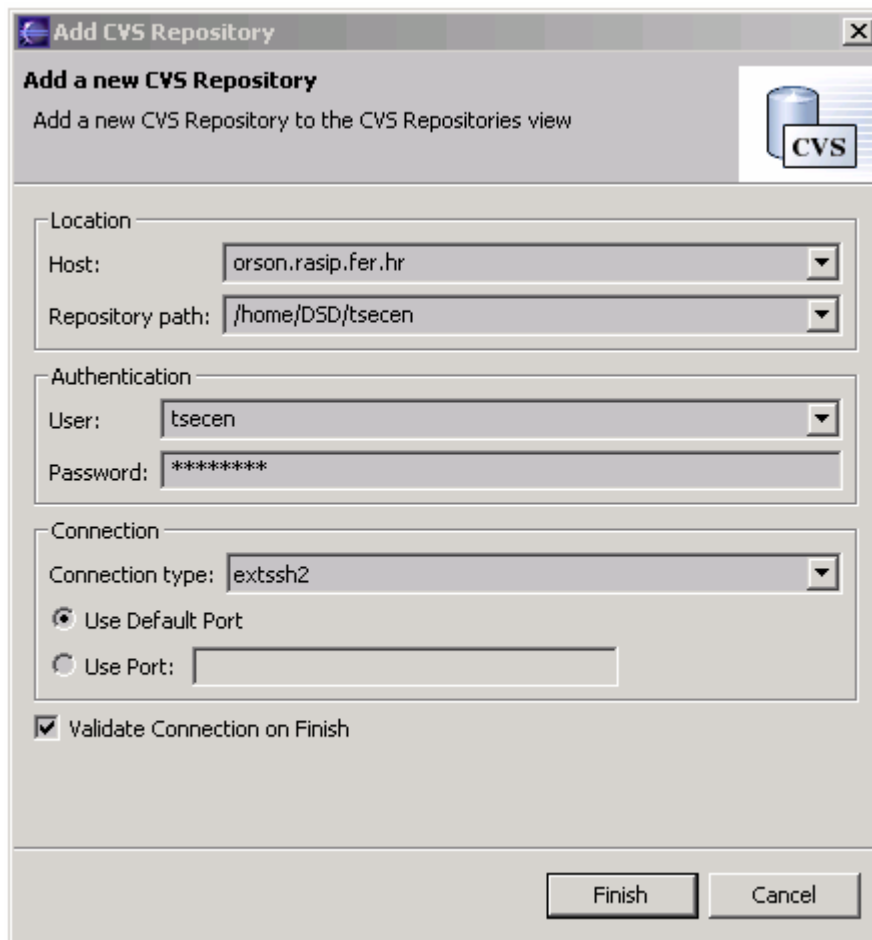


Fig. F.1: Adding the CVS repository directory

Click Finish. The repository location is created.

Sharing a new project using CVS

There are several scenarios that can occur when sharing a project using CVS. The most common scenario is sharing a new project using CVS when the project does not already exist remotely.

To share a new project using CVS:

In the Navigator view, select the project to be shared

Select **Team -> Share Project...** from the project's pop-up menu. The Share Project wizard opens.

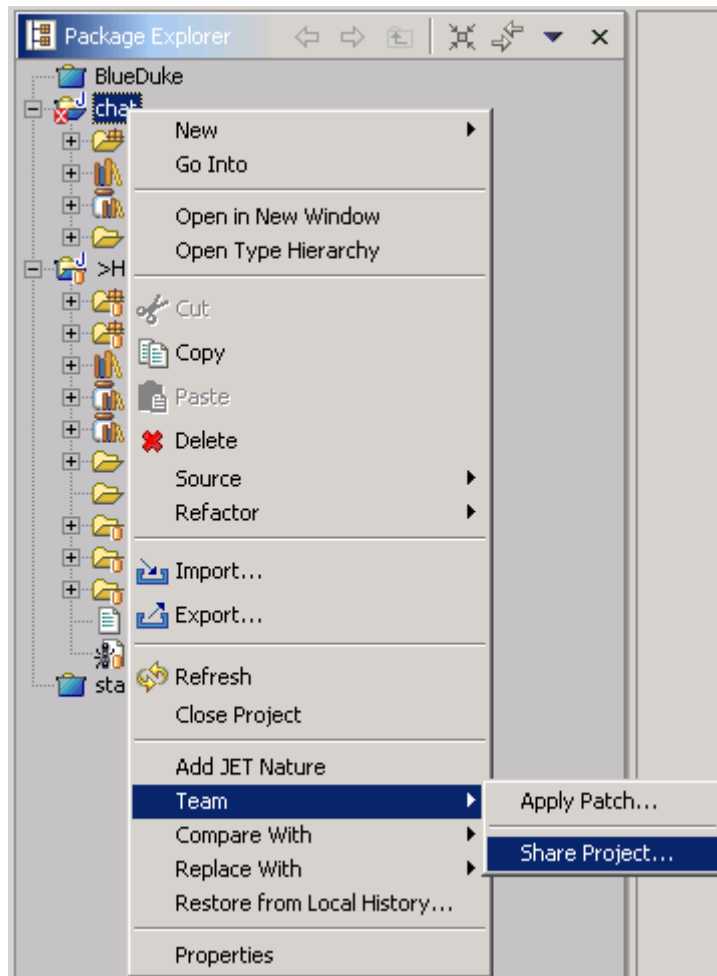


Fig. F.2: Sharing a project using CVS

From the list of available repository providers, choose **CVS** and click **Next** to go to the next page. (**Note:** If there is only one repository provider available, the first page will be skipped and next page will be displayed automatically.)

Select the target repository from the list of known repositories or, if the target repository is not in this list, choose to create a new repository location and click **Next**.

If entering a new repository location, enter the repository information and click **Next** when completed. (**Note:** this page is the same format as the [Creating a CVS repository location](#) wizard.)

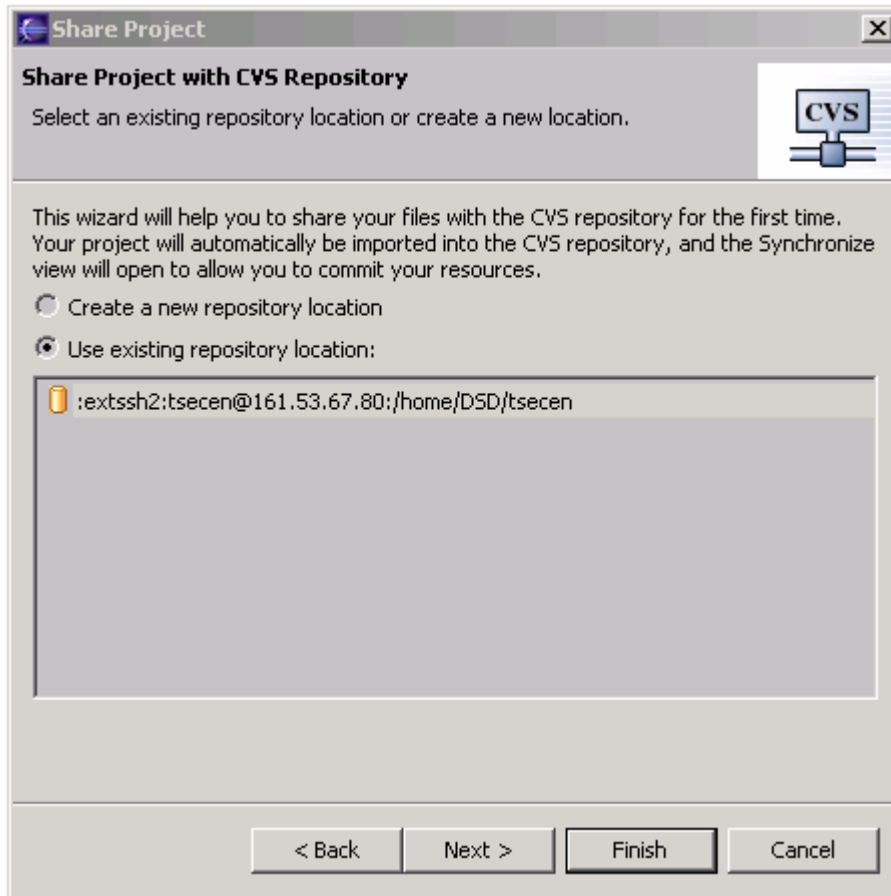


Fig. F.3: Sharing an existing CVS repository

(Optional) Either choose to use the name of the local project as the remote project name or enter another name.

Click **Finish** to share the project with the repository. The project folder will be created remotely and the Synchronize view will open and allow the committing of the project's resources. (**Note:** If the project already exists remotely, the Synchronize view will show conflicts on any files that exist both locally and remotely.)

Checking out a project from a CVS repository

To check out a project from a CVS repository to the Workbench:

1. Switch to the CVS Repository Exploring perspective or add the CVS Repositories view to the current perspective.
2. In the CVS Repositories view, expand the repository location.
3. Expand **HEAD** and select the folders that you want to add as projects to the Workbench. If you are looking for a folder in a particular version:
 - a. Expand the **Versions** category and find the folder name that you want to add to the Workbench.
 - b. Expand the folder to reveal its versions.

If you are looking for the latest folder in a branch:

- c. Expand the **Branches** category.
 - d. Expand the branch that holds the folder that you want to add to the Workbench.
4. From the pop-up menu for the selected folders, select one of the following:

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

- a. **Check Out as Project** to check out each of the selected folders as a project in the local workspace with the same name as the folder in the repository.
- b. **Check Out As...** to check out the selected folders into a custom configured project in the local workspace. *Note:* When multiple folders are selected, this operation only allows the specification of a custom parent location for the new projects.

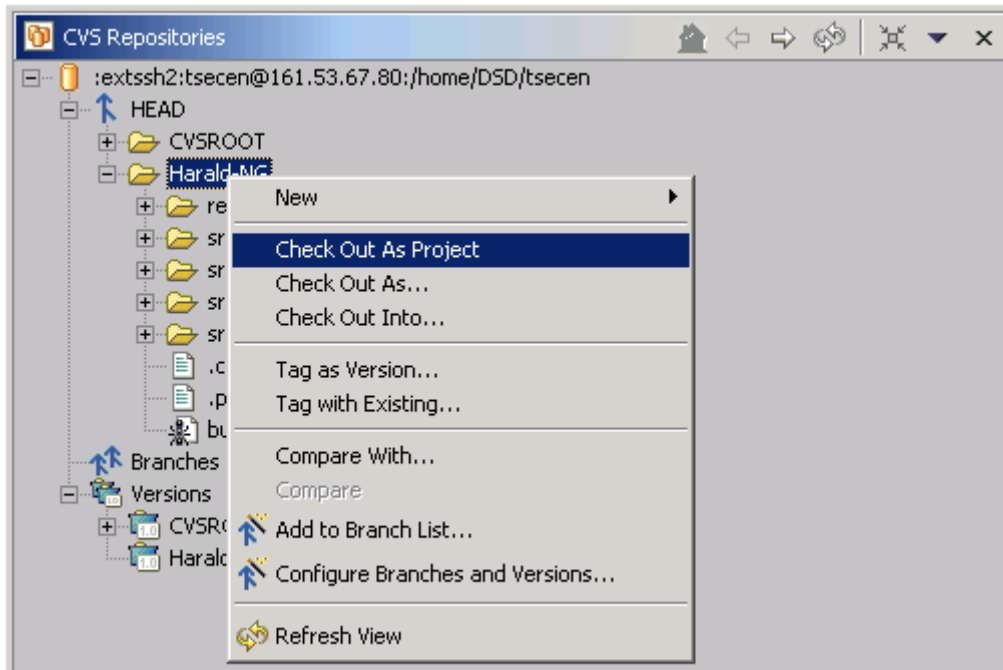


Fig. F.4: Creating a local project from CVS repository

5. If **Check Out As...** was chosen on a single project, one of two possible dialogs is displayed depending on whether the folder in the repository contains a `.project` file or not.
 - a. If there is a `.project` file, the dialog will accept a custom project name and location.
 - b. Otherwise, the dialog will be the New Project wizard that allows full customization of the project (e.g. Java project).

Tip: Any folder, including non-root folders, can be checked out from a CVS repository.

Updating

While you are working on a project in the Workbench, other members of your team may be committing changes to the copy of the project in the repository. To get these changes, you may "update" your Workbench to match the state of the branch. The changes you will see will be specific to the branch that your Workbench project is configured to share. You control when you choose to update.

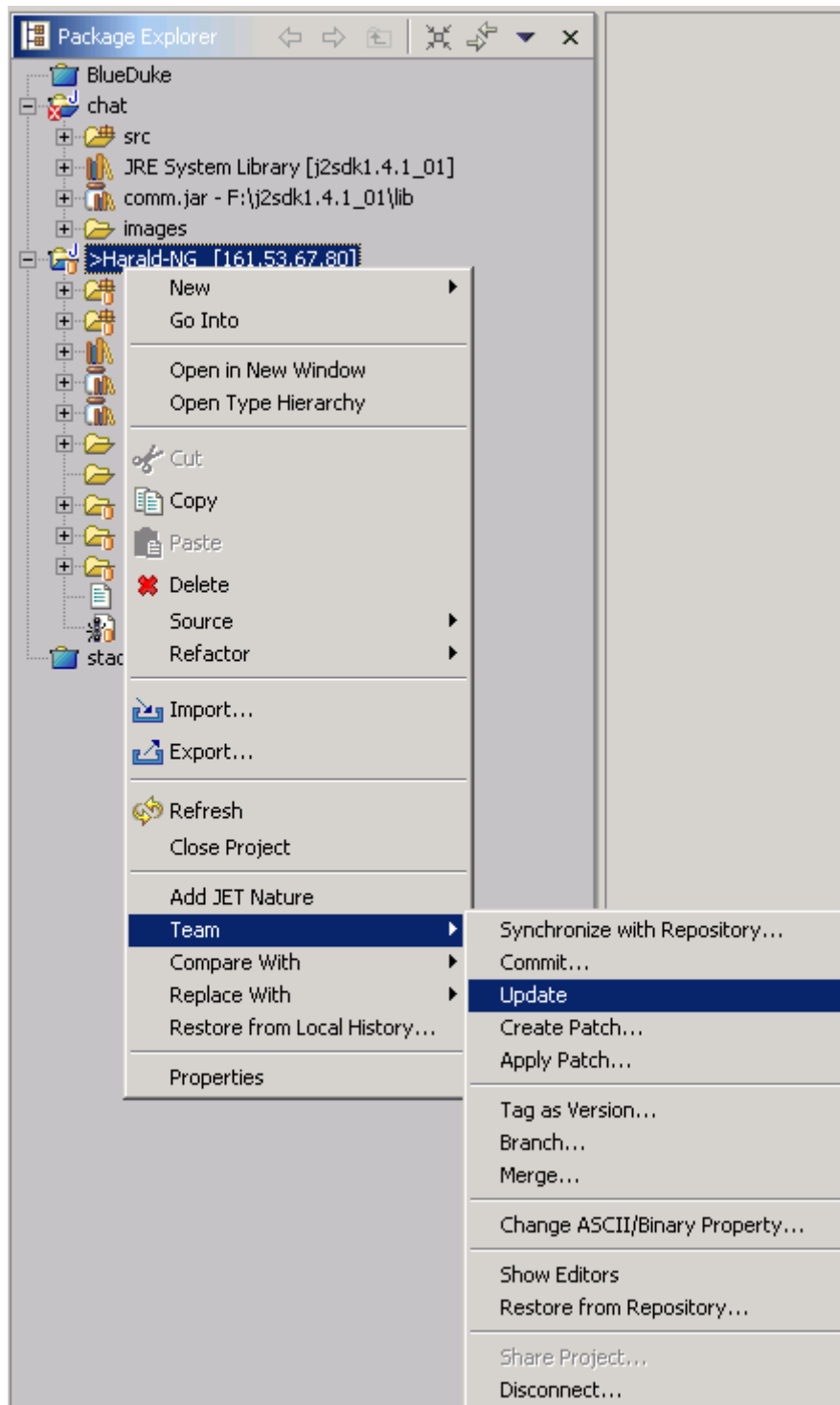


Fig. F.5: Updating the local project

The update command can be issued from two places: the **Team > Update** menu, or the **Synchronize** view. In order to understand the difference between these two commands, it is important to know about the three different kinds of incoming changes.

- A *non-conflicting* change occurs when a file has been changed remotely but has not been modified locally.
- An *automergable conflicting* change occurs when an ASCII file has been changed both remotely and locally (i.e. has non-committed local changes) but the changes are on different lines.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

- A *non-automergable conflicting* change occurs when one or more of the same lines of an ASCII file or when a binary file has been changed both remotely and locally (binary files are never automergable).

When you select **Team > Update**, the contents of the local resources will be updated with incoming changes of all of the above three types. For non-conflicting and automergable conflicts, there is no additional action required (for automergable conflicts, the changed local resource is moved to a file prefixed with ".#" just in case the automerge wasn't what the user wanted). However, for non-automergable conflicts, the conflicts are either merged into the local resource using special CVS specific markup text (for ASCII files) or the changed local resource is moved to a file prefixed with ".#" (for binary files). This matches the CVS command line behavior but can be problematic when combined with the Eclipse auto-build mechanism. Also, it is often desirable to know what incoming changes there are before updating any local resources. These issues are addressed by the Synchronize view.

To open the Synchronize view in incoming mode:

1. In the Navigator view, select the resources that you want to update.
2. From the pop-up menu for the selected resources, select **Team > Synchronize with Repository**. The Synchronize view will open.
3. On the toolbar of the Synchronize View, click the **incoming mode** button to filter out any modified Workbench resources (outgoing changes) that you may have.

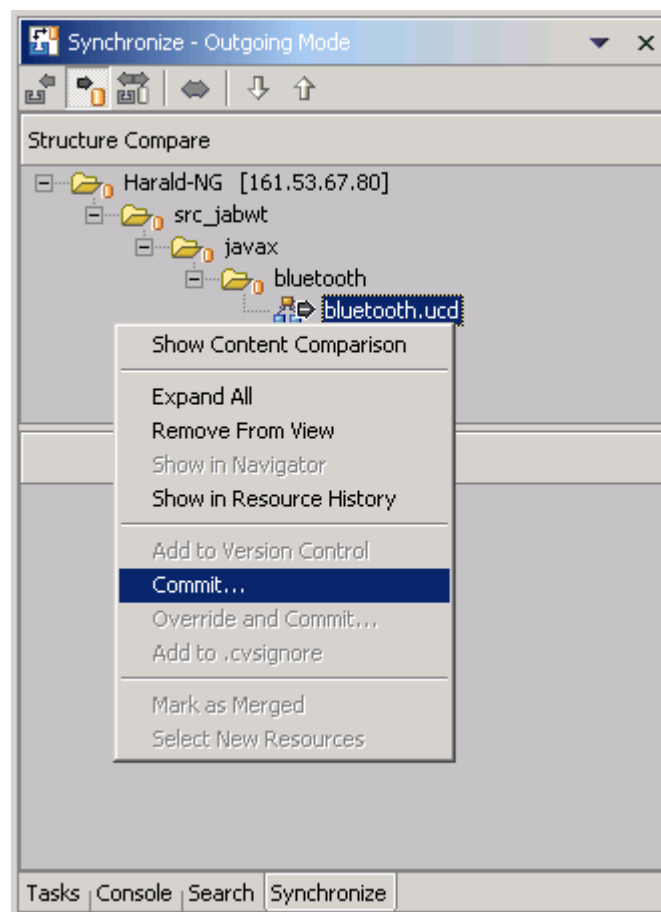


Fig. F.6: Committing the local changes

In incoming mode, you will see changes that have been committed to the branch since you last updated. The view will indicate the type of each incoming change (non-conflict, automergable

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

conflict or non-automergable conflict). There are two update commands (available from the context menu of any resource in the view) to deal with the different types of conflicts: **Update from Repository** and **Override and Update**. When you select the **Update from Repository** command in the Synchronize view, only non-conflicting changes are processed, leaving any files that have automergable or non-automergable conflicts in the view (any files that have been successfully processed are removed from the view). The **Override and Update** command operates on the two types of conflicts. After selecting this command, you will be prompted before a merge is attempted and asked if you want to auto merge the contents or overwrite them with the repository file. If you select to auto merge then only automergable conflicts will be processed and the incoming changes will be automerged with the local changes. Otherwise all conflicts will be processed and the local resources will be replaced with the remote contents. This "replace" behavior is rarely what is desired. An alternative is described later.

To update non-conflicting and automergable files:

1. The Structure Compare pane at the top of the Synchronize view contains the hierarchy of resources with incoming changes.
2. Select the non-conflicting files and choose **Update from Repository** from the pop-up menu. This will update the selected resources and remove them from the view.
3. Select the automergable conflicts and choose **Override and Update** from the pop-up menu. Select to only update automergable resources and click OK when prompted. This will update the selected resources and remove them from the view.

If your local Workbench contains any outgoing changes that are not automergable with incoming changes from the branch, then, instead of performing an **Override and Update**, you can merge the differences into your Workbench manually, as follows:

1. In the Structure Compare pane, if there is a conflict in the resource list (represented by red arrows), select it.
2. In the Text Compare area of the Synchronize view, local Workbench data is represented on the left, and repository branch data is represented on the right. Examine the differences between the two.
3. Use the text compare area to merge any changes. You can copy changes from the repository revision of the file to the Workbench copy of the file and save the merged Workbench file (using the pop-up menu in the left pane).
4. Once you are completed merging the remote changes into a local file, choose **Mark as Merged** from the pop-up menu. This will mark the local file as having been updated and allow your changes to be committed.

Note: The repository contents are not changed when you update. When you accept incoming changes, these changes are applied to your Workbench. The repository is only changed when you commit your outgoing changes.

Tip: In the Structure Compare pane, selecting an ancestor of a set of incoming changes will perform the operation on all the appropriate children. For instance, selecting the top-most folder and choosing **Update from Repository** will process all non-conflicting incoming changes and leave all other incoming changes unprocessed.

Warning: The behavior of the **Override and Update** command described above only applies to the incoming mode of the Synchronize view. In the **incoming/outgoing mode** of the view, the behavior for incoming changes and conflicts is the same but the outgoing changes are reverted by the commands to whatever the repository contents are. Exercise great caution if using this command in incoming/outgoing mode.

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Committing

You can commit Workbench resources that you have modified to the repository so that other team members can see your work. Only those changes committed on that branch will be visible to others working on that branch. The commit command can be issued from two places: the **Team > Commit** menu, or the **Synchronize** view.

To commit changes using **Team > Commit**:

1. In the Navigator view, select the resources that you want to commit.
2. Right-click on the resources and select **Team > Commit** from the pop-up menu.
3. In the Commit Comment dialog box, provide a comment for your changes (for example, Fixed the spelling mistakes).

If there are conflicting changes on any files that are committed in the above fashion, the operation will fail. If this occurs, you must either perform an update or use the Synchronize view to resolve the conflicts. It is considered a more ideal workflow to always update before committing in order to ensure that you have the latest state of the repository before committing more changes.

If one or more of the resources being committed are new and not yet added to CVS control, you will be prompted to add these resources before the commit is performed. You can choose to add all, some or none of the new resources to CVS control before performing the commit. Any resources that are not under CVS control will not be committed. Committing from the Synchronize view also prompts if there are new resources.

To commit changes in the Synchronize view:

1. In the Navigator view, select the resources that you want to commit.
2. Right-click to open the pop-up menu and select **Team > Synchronize with Repository**. The Synchronize view will open.
3. On the toolbar of the Synchronize view, select the **outgoing mode** button to show any modified Workbench resources (outgoing changes) that you may have.
4. If there are conflicts (red arrows), resolve them. Use the text compare area to merge resources with conflicts. You can copy changes from the repository revision of the file to the Workbench revision of the file and save the merged Workbench resource. Once all the conflicts in the Structure Compare area have been resolved, you are ready to commit.
5. In the Structure Compare pane, right-click the top of the hierarchy that you want to commit, and select **Commit** from the pop-up menu.
6. In the Commit Comment dialog box,

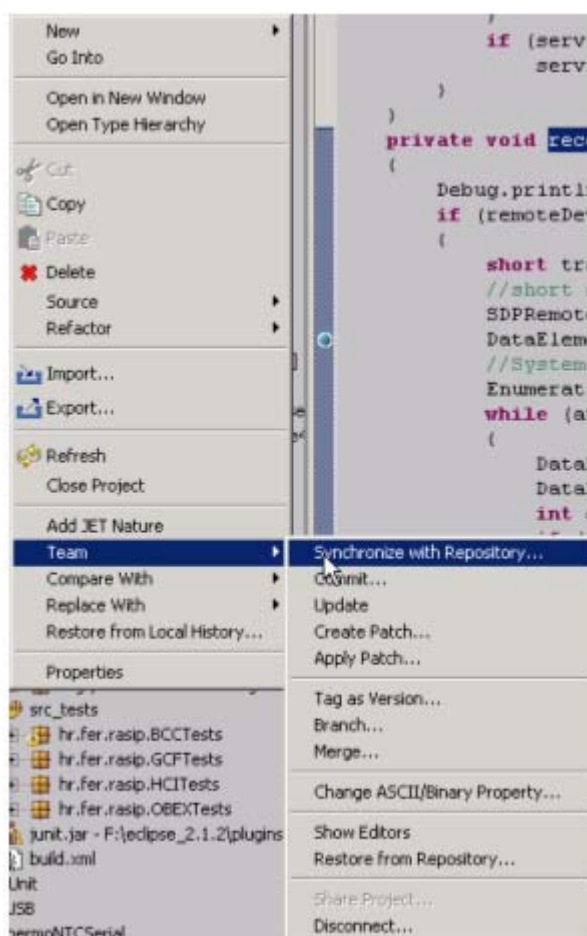


Fig. F.7: Synchronizing project with the repository

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

provide a comment for your changes (for example, Fixed the spelling mistakes).

Tip: You can commit files that are in conflict by performing an **Override and Commit**. This will commit the Workbench copy of the resource into the repository and thus remove any of the incoming changes.

Warning: The behavior of the **Override and Commit** command described above only applies to the outgoing mode of the Synchronize view. In the **incoming/outgoing mode** of the view, the behavior for outgoing changes and conflicts is the same but the incoming changes are reverted by the commands to whatever the local Workbench contents are. Exercise great caution if using this command in incoming/outgoing mode.

Synchronizing with a CVS repository

In the CVS team-programming environment, there are two distinct processes involved in synchronizing resources: *updating* with the latest changes from a branch and *committing* to the branch.

When you make changes in the Workbench, the resources are saved locally. Eventually you will want to commit your changes to the branch so others can have access to them. Meanwhile, others may have committed changes to the branch. You will want to update your Workbench resources with their changes.

Important! It is preferable to update *before* committing, in case there are conflicts with the resources in your Workbench and the resources currently in the branch.

The synchronize view contains filters to control whether you want to view only *incoming changes* or *outgoing changes*. Incoming changes come from the branch. If accepted, they will update the Workbench resource to the latest version currently committed into the branch. Outgoing changes come from the Workbench. If committed, they will change the branch resources to match those currently present in the Workbench.

Regardless of which mode (filter) you select, the Synchronize view always shows you conflicts that arise when you have locally modified a resource for which a more recent version is available in the branch. In this situation you can choose to do one of three things: update the resource from the branch, commit your version of the resource to the branch, or merge your work with the changes in the branch resource. Typically you will want to merge, as the other two options will result in loss of work.

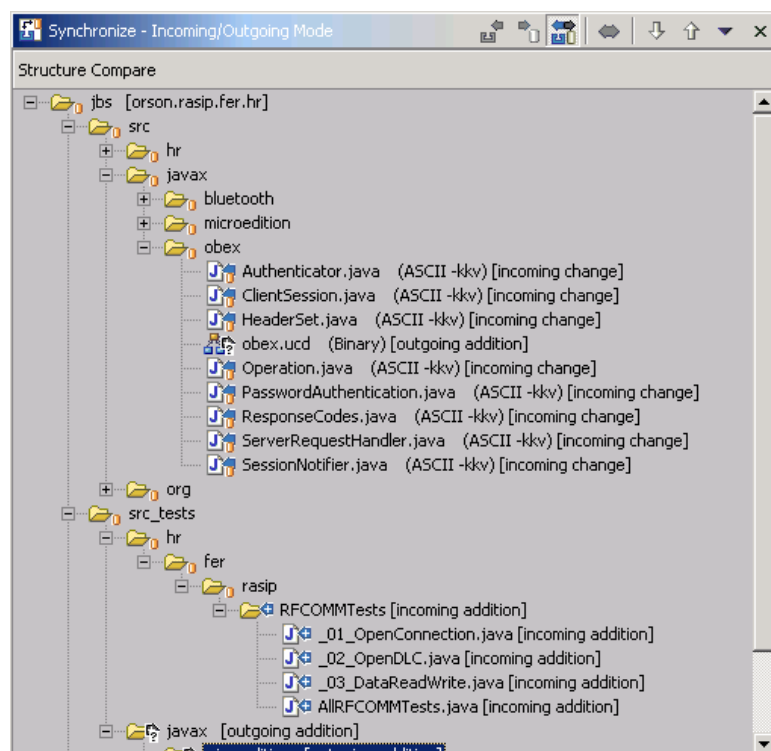


Fig. F.1: Synchronizing view

Java Bluetooth stack	Version: 1.0
Developer's Guide	Date: 2004-01-15

Literature and resources

Books

- [1] B. Hopkins, R. Anthony: *Bluetooth for Java*, Apress 2003.
- [2] Bluetooth SIG: *Specification of the Bluetooth System v1.1, vol. 1*, Bluetooth SIG, February 2001.

Links on the Internet

- [1] <http://sourceforge.net/projects/javablueetooth> - Javablueetooth on SourceForge
- [2] <http://bluez.sourceforge.net/> - BlueZ - Bluetooth stack for Linux
- [3] <http://java.sun.com/products/javacomm/downloads/index.html> - commAPI download
- [4] <http://www.eclipse.org> – Eclipse platform homepage
- [5] <http://www.junit.org> – JUnit homepage
- [6] [http://kobjects.dyndns.org/kobjects/auto?self=\\$81d91ea1000000f5d0738100](http://kobjects.dyndns.org/kobjects/auto?self=$81d91ea1000000f5d0738100) – ME4SE homepage
- [7] <http://www.sixlegs.com/software/png/> - sixlegs PNG library (classpath necessity for ME4SE)
- [8] www.jcraft.com/eclipse-cvsssh2/ - Eclipse CVSssh2 plug-in
- [9] <http://java.sun.com/products/j2mewtoolkit/> - Sun Wireless Toolkit
- [10] <http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth1/> - deploying Wireless Java applications
- [11] <http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth2/> - Wireless Application programming with J2ME and Bluetooth
- [12] <http://developers.sun.com/techtopics/mobility/midp/articles/threading2/> - using threads in J2ME applications
- [13] <http://jbs.sourceforge.net/> - JavaBluetoothStack on SourceForge.net
- [14] www.jcp.org/jsr/detail/82.jsp - JSR-82 specification page

Index