NDGate
User's Manual

Version 1.0

January 2008

Contents

1 Revision **7**

2 Description **8**

3 Installation **9**

3.1 Requirements 9

3.2 Installation 9

3.2.1 Development environment 9

3.2.2 Bluetooth dongle 10

3.2.3 Bluetooth environment 10

3.2.4 Internet 10

3.3 Application configuration 11

3.3.1 Variables 12

4 Accessing the API **13**

4.1 UDP-connections 13

4.2 A command 13

4.3 General structures and need to know 15

4.3.1 Device address 15

4.3.2 Services 16

4.3.3 Phone book storages 16

4.3.4 List/storage entry type 17

4.3.5 Phone number 17

4.3.6 SMS storages 17

4.3.7 SMS storage types 17

4.3.8 SMS satus 18

4.3.9 Encoding 18

4.3.10 Battery status 19

4.3.11 Multiparty actions 19

4.3.12 Events 19

4.3.13 Call status 20

4.3.14 Audio for headset and handsfree 20

4.3.14.1 Using WaveIn and WaveOut 21

4.3.14.2 Using btAudioClient.exe 21

4.3.15 Error codes 22

4.3.16 Sub-error codes 23

5 Finder commands- Device/Service search and bond/unbond 24

5.1 Device finding/handling commands . . . 24

5.1.1 startDeviceSearch - Start searching for available devices . . . 24

5.1.2 stopDeviceSearch - Stop searching for available devices . . . 24

5.1.3 getDeviceSearchStatus - Get status for device search . . . 25

5.1.4 getServices - Get services for a device . . . 25

5.1.5 bond - Bond a device . . . 25

5.1.6 unbond - Unbond a device . . . 26

5.1.7 ping - Ping the gateway . . . 26

5.1.8 getSupportedServices - Get the supported gateway services . . . 27

5.2 Incomming unsolicited commands . . . 27

5.2.1 newDevice - New device found during device search . . . 27

6 General commands for the serial, headset, handsfree, avrcp, obexftp, syncml, avrcp and a2dp service 28

6.1 Connection commands . . . 28

6.1.1 connect - Connect to a device . . . 28

6.1.2 disconnect - Disconnect from a device . . . 29

6.1.3 getStatus - Method name short description . . . 29

7 Commands for serial, handsfree and headset service 30

7.1 Phone book specific commands . . . 30

7.1.1 getLists - Get available phone book storages . . . 30

7.1.2 getListParams - Get parameters for a phone book storage . . . 30

7.1.3 getList - Get all entries in a list/ from a storage . . . 31

7.2 Device functionality commands . . . 32

7.2.1 getEncoding - Get the current encoding for received data . . . 32

7.2.2 getBatteryStatus - Get battery level and status . . . 32

7.2.3 getSignalQuality - Get the signal quality . . . 33

7.2.4 getValidRingVolume - Get valid ring volume interval . . . 33

7.2.5 setRingVolume - Set the ring volume . . . 33

7.2.6 getRingVolume - Get the current ring volume . . . 34

7.2.7 setSilence - Set silence mode . . . 34

7.2.8 getSilence - Get silence mode status . . . 35

7.2.9 setKeypressNotification - Enable or disable key press notification . . . 35

7.3 Call specific commands . . . 35

7.3.1 dial - Dial a number . . . 35

7.3.2 answer - Answer an incoming call . . . 36

7.3.3 hangup - Hangup an ongoing or incoming call . . . 36

7.4 SMS specific commands . . . 36

7.4.1 sendSMS - Create and send a new SMS . . . 36

7.4.2 getSMSStorages - Get SMS storages for a storage type . . . 37

7.4.3 setSMSStorage - Set the storage for a SMS storage type . . . 37

7.4.4 readSMS - Read a SMS from a storage . . . 38

7.4.5 getSMSList - Get all SMSes in a storage . . . 38

7.4.6 setSMSNotification - Enable or disable SMS notification . . . 39

7.5 Incomming unsolicited commands . . . 39

7.5.1 newSMS - New SMS received . . . 39

7.5.2 ring - New incoming call 40
7.5.3 number - Incoming number 40
7.5.4 keypress - A key has been pressed 40

8 Commands for the headset, handsfree and a2dp service 41
8.1 Incomming unsolicited commands 41
8.1.1 audioEvent - Audio status changed 41

9 Commands for the handsfree service 42
9.1 Call specific commands 42
9.1.1 getSupportedMultiparty - Get available call hold and three way actions 42
9.1.2 doMultipartyAction - Perform a multiparty action 42
9.1.3 getCallStatus - Get status for all calls 43
9.1.4 sendDTMF - Send DTMF 43
9.2 Device functionallity commands 44
9.2.1 getAvailableEvents - Get available events 44
9.2.2 getEventValue - Get current value for an event 44
9.2.3 getNetworkOperator - Get the name of the network operator 44
9.3 Incoming unsolicited commands 45
9.3.1 event - New event 45
9.3.2 callWaiting - New waiting incoming call 45

10 Commands for the dun service 46
10.1 Connection commands 46
10.1.1 connect - Connect to a device 46
10.1.2 disconnect - Disconnect from a device 46
10.1.3 getStatus - Method name short description 46

11 Scenarios and usages 47
11.1 Parsing and creating a command 47
11.2 Find and automatically connect to a device 47
11.3 Reading phone book entries 48
11.4 Reading SMSes 48
11.5 Call handling for the handfree service 49
11.5.1 Placing a call 49
11.5.2 Receiving a call 49
11.5.3 Multiparty calls 49

12 Troubleshooting 51
12.1 Problems and solutions 51
12.1.1 How to update a bug in this manual? 51
12.1.2 Error messages 51
12.1.2.1 Invalid argument device. The device cannot be found. 51
12.1.2.2 Another audio connection is active to the requested remote device. 51
12.1.2.3 An unknown or internal error occured./Unknown error. 51
12.1.2.4 Runtime errors 52
12.1.2.5 No UDP data is received when using infoTainer. 52
12.1.2.6 Audio routing does not work. 52

13 Wordlist 53

14 Appendix **54**

14.1 Handsfree workaround 54

14.2 C# command parsing class 56

14.2.1 File btCommand.cs 56

14.3 C++ command parsing class 58

14.3.1 File btCommand.h 58

14.3.2 File btCommand.cpp 59

14.4 C++ audio streaming 61

14.4.1 File audio.h 61

14.4.2 File audio.cpp 62

14.4.3 File main.cpp 68

List of Figures

2.1 System layout 8

14.1 Spylite configuration 55
14.2 Handsfree service spylite sequence entries 56

List of Tables

1.1 Revision history . . . 7

3.1 Configuration variables . . . 12

4.1 UDP port mapping . . . 13
4.2 One or more arguments . . . 14
4.3 Zero arguments . . . 14
4.4 Command examples . . . 14
4.5 Error command . . . 15
4.6 Services . . . 16
4.7 Phonebook storages . . . 16
4.8 List entry types . . . 17
4.9 SMS storages . . . 17
4.10 SMS storage types . . . 18
4.11 SMS statuses . . . 18
4.12 Encodings . . . 18
4.13 Battery status . . . 19
4.14 Multiparty actions . . . 19
4.15 Events . . . 20
4.16 Call statuses . . . 20
4.17 WaveIn/waveOut variables . . . 21
4.18 Error codes . . . 22
4.19 Sub-error codes . . . 23

Chapter 1

Revision

Version	***Date***	***Name***	***Comments***
1.0	January 2008	Johan Böhlin	Initial document.

Table 1.1: Revision history

Chapter 2

Description

NDGate provides an API over UDP for easy access of functionalities available on a nomadic device like a mobile phone. The nomadic device is connected wireless using Bluetooth to the gateway. Typical scenario is an infotainment system in a car that wants to integrate with the drivers mobile phone.

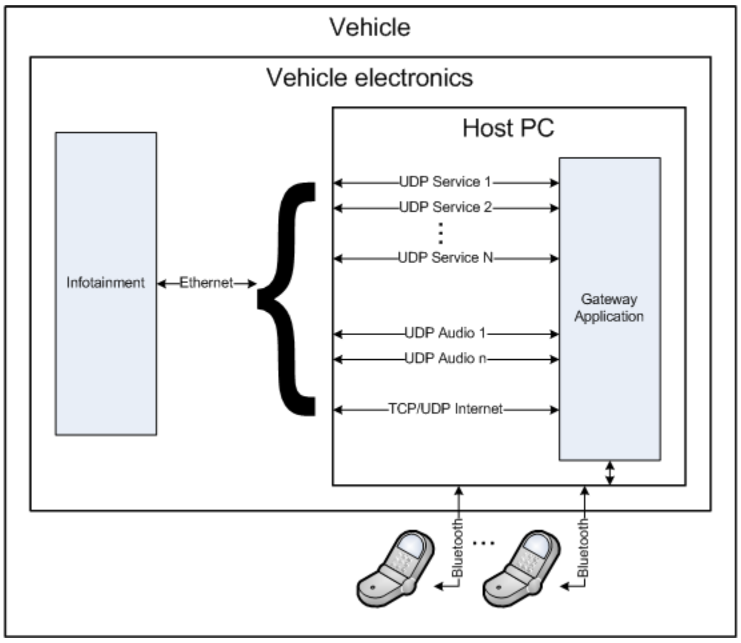


Figure 2.1: System layout

Chapter 3

Installation

3.1 Requirements

The gateway has been tested on two different windows XP installations, so no other operative systems are officially supported. This is what you need:

- Windows XP with Service pack 2
- A Bluetooth USB dongle that supports the Broadcom/Widcomm Bluetooth-stack. The gateway is tested with the D-link DBT-122 Bluetooth-dongle.

3.2 Installation

3.2.1 Development environment

You need to install the following items:

1. Microsoft Windows Server 2003 R2 Platform SDK, `http://www.microsoft.com/downloads/details.aspx?FamilyId=0BAF2B35-C656-4969-ACE8-E4C0C0716ADB&displaylang=en`
2. Microsoft Visual C++ 2005 Redistributable Package (x86), `http://www.microsoft.com/downloads/details.aspx?familyid=32bc1bee-a3f9-4c13-9c99-220b62a191ee&displaylang=en`
3. Microsoft Visual Studio C++ Express 2008, `http://www.microsoft.com/express/download/`
4. If you are going to use the test application infoTainer, you have to install Visual Studio C# express 2008 from the same url.
5. Broadcom Bluetooth SDK Version 6.1.0.1501, `http://www.broadcom.com/products/bluetooth_sdk.php`

You also need to copy btwapi.dll from your "WIDCOMM\BTW DK\SDK\Release" folder to your "%SYSTEMROOT%\system32" directory.

To build NDGate, you need to configure your Visual C++ paths. Open Visual C++ Express 2008 and goto the menu Tools->Options->Projects and Solutions->VC++ Directories.
For "Executable files" add search path to your "Microsoft Platform SDK for Windows Server 2003 R2\Bin".
For "Include files" add search path to your "Microsoft Platform SDK for Windows Server 2003 R2\Include" and your "WIDCOMM\BTW DK\SDK\Inc".

For ”Library files” add search path to your ”Microsoft Platform SDK for Windows Server 2003 R2\Lib” and your ”WIDCOMM\BTW DK\SDK\Release”.
Note that you need to find the correct search path for **your** installation on your computer.

3.2.2 Bluetooth dongle

Install your Bluetooth dongle using the drives you got with your dongle. They need to be broadcom/widcomm drivers. **Important!** You need to disable all other Bluetooth dongles! Only one dongle can be enabled, and it must use the widcomm/broadcom drives.

3.2.3 Bluetooth environment

These steps has to be done in the Windows environment.
First we need to disable the local headset and auto gateway services.

1. Open ”My Bluetooth places” in ”My Computer” or Go to ”control panel”.
2. Open ”My device”.
3. Open ”My services”.
4. Find ”Headset”.
5. Right click on ”Headset” and chose properties.
6. Uncheck ”auto connect”.
7. Repeat step 4 to 6 for ”Audio Gateway”.

3.2.4 Internet

There are a couple of steps needed for the internet service. First we need to enable the Bluetooth-modem. This may not be necessary it the Bluetooth-modem is already installed.

1. First, you’ll need a random mobile phone with Bluetooth DUN support.
2. Enable service discovery on the mobile phone, and search for the device in windows using ”My Bluetooth places”, usually located in ”My Computer” -> ”My Bluetooth places” -> ”Search for Bluetooth devices”. When the device is found, double-click on it to search for services. Once all services are found, right click on the ”dial-up networking” service chose ”connect...”
3. Pair the mobile phone if requested.
4. Once connected, the ”found new hardware” dialog or similar should pop-up. Go trough the modem installation. Once the modem is installed, you should disconnect the Bluetooth connection to the mobile phone.
5. Go to windows ”Control panel” -> ”Phone and Modem Options” -> ”Modem” - tag and verify that a Bluetooth modem is installed. Also note the COM-port number the modem is connected to, you need this in the next step. If not, go to step 1 and retry again. You can also try another mobile phone.
6. **This is important!** Set the bluetoothModemComPort setting in settings.ini (see section 3.3) to the COM-port number you got from the previous step. Now we need to create a dial-up connection.

7. Goto "Control panel" and then "Network connections"
8. Remove ALL **Dial-up** connections (not LAN or high-speed internet), if there are any.
9. Go trough the "New Connection Wizard".
10. Choose "connect to internet" and next.
11. "set up my connection manually" and next.
12. In the next step, chose "Connect using dial-up modem".
13. **This is important!** Type in exactly this name as ISP name: *bluetooth*
14. Leave phone number blank.
15. Create the connection for "Anyone's use".
16. Leave the username and password blank.
17. Check "use this account name and..." and "default connection"
18. When the "Connect Bluetooth" dialog appears, close it.
19. Right click on the newly created connection, click on "Properties".
20. On the "general" tab, make sure the "Bluetooth-modem" is selected.
21. Goto the "Options" tab. Uncheck "Display progress while connecting, "Promt for name and password,..." and "Prompt for phone number".
22. This and the next step is only required if you need internet on another computer than where the NDGate.exe is running. Goto the "Advanced" tab. Check "Allow other network users to connect trough this computer's internet connection". If you receive a dialog saying "The username and password for..", just click OK.
23. Select the network interface card in the in "Home networking connection"-dropdown that should be allowed to use the internet, i.e. the network card connected to the same network where the client/GUI application is connected to.
24. Click OK to save the settings.

3.3 Application configuration

At start-up, NDGate.exe to read a settings file, settings.ini, from the same directory NDGate.exe is started from. If the file does not exist, or the settings key is missing in the settings.ini-file, the default value is used. Create a new file named settings.ini with the content below, and place the file n the same directory as NDGate.exe:

```
[NDGate]
udpBase=9000
udpDestinationHost=127.0.0.1
bluetoothModemComPort=4
logToFile=1
logToConsole=1
```

3.3.1 Variables

Variable	**Values**	**Default**	**Description**
udpBase	1-65535	9000	*The base UDP-port. See section 4.1*
udpDestinationHost	a string	127.0.0.1	*The destination address where UDP-packets are sent to from the gateway.*
bluetoothModemComPort	1-30	3	*The COM-port number the Bluetooth modem is connected to. This setting must be set! See section 3.2.4.*
logToFile	0-1	0	*Prints log messages to NDGate.log in the NDGate.exe directory. 0 to disable.*
logToConsole	0-1	1	*Prints log messages to console window. 0 to disable.*
handsfreeDevices	devices		*Device addresses on the format specified in section 4.3.1, separated by comma (no space).*
handsfreeScns	Scns		*The handsfree SCN-port number for each device in handsfreeDevices. See appendix 14.1 about the handsfree workaround.*

Table 3.1: Configuration variables

Chapter 4

Accessing the API

4.1 UDP-connections

The API is accessed by UDP connections. Each service has its own UDP-port-pair. The client should establish an UDP-listener to each service that the client wants to use.

The base address is configured in the settings.ini file. See section 3.3

Service	**Out to gateway**	**In from gateway**
finder	Base+1	Base+0
serial	Base+3	Base+2
headset	Base+5	Base+4
obexftp	Base+7	Base+6
syncml	Base+9	Base+8
avrcp	Base+11	Base+10
a2dp	Base+13	Base+12
dun	Base+15	Base+14
handsfree	Base+17	Base+16
Headset-audio	Base+101	Base+100
Handsfree-audio	Base+103	Base+102
A2DP-audio	Base+105	Base+104

Table 4.1: UDP port mapping

The client should send commands/data on the ”Out to gateway”-port, and receive responses/data on the ”In from gateway”-port.
A service can only be connected to one device. It possible to multiple devices connected to the gateway, using different services.

4.2 A command

A command has three parts: A function name, zero or more arguments and a terminator.

The command is formatted as one of the two follow ways:

For one or more arguments:

Function name	<space>	One or more arguments	terminator

Table 4.2: One or more arguments

For zero arguments:

Function name	terminator

Table 4.3: Zero arguments

The *function name* is the name of the function. Valid characters for a function name are the letters a-z, A-Z and 1-9. Space, tab, new line etc. are invalid characters in a function name.
If there are no arguments, the terminator is followed immediately after the function name.

The *terminator* consists of the carrier return- followed by the new line character, which is written \r\n, which corresponds to the ASCII values 13 and 10 respectively.

If there is at least one argument, a space separates the function name and the arguments. The first character after the space is part of the first (or only) argument. Each argument is separated with the character sequence **{:}**, that is a left frizz-parenthesis, a colon and finally a right frizz-parenthesis. An argument can be zero or more characters. The last argument is ended with the *terminator.* An argument cannot contain the **{:}** sequence.

Command examples:

functionName argument1{:}argument2{:}argument3\r\n	A command with three arguments.
functionName argument1\r\n	A command with one argument.
functionName argument1{:}{:}argument3\r\n	A command with three arguments, where argument2 is empty.
functionName\r\n	A command with no argument.
functionName \r\n	A command with one empty argument.
functionName {:}\r\n	A command with two empty arguments.
functionName {:}{:}\r\n	A command with three empty arguments.
<space>functionName\r\n	An **INVALID** command.
\r\n	An **INVALID** command.

Table 4.4: Command examples

Command responses

All command sent are responded with the same command structure, and with zero or more arguments.

The function name for a response command has the string *Return* appended to the function name. Example: If the command *functionName argument1*\r\n are sent, a possible return argument could be *functionNameReturn\r\n* or *functionNameReturn argument1\r\n*.

Error responses Responses as described above are only sent in case of no error. If an error occurs during a command, an error response is sent instead. The error command are formatted as in table 4.5.

error	terminator	<space>	errValue{:}subErrValue{:}errStr	terminator

Table 4.5: Error command

Where the function name is *error*, and argument 1 is an string and one of the values in the error listing, see section 4.3.15, argument 2 is an string and one of the values in the Sub-Error listing, see section 4.3.16, and errStr is a human readable string of the error occurred, useful for debugging. An example of an error response:
error errInvalidArguments{*:*}*subErrInvalidNumber*{*:*}*The number is invalid.*\r\n

Lists
An argument could be a list. A list separate its elements using the | character (ASCII = 124).
Example lists:

functionNameReturn element1|*element2*|*element3*||*element5:argument2*\r\n

Where the list is *element1*|*element2*|*element3*||*element5* with five elements, where element 4 is empty.

4.3 General structures and need to know

4.3.1 Device address

A device address is a string hexadecimal representation of a device. The address if separated in six bytes using a colon as separator. The format is as follows: byte5:byte4:byte3:byte2:byte1:byte0
An example of an address: *1b:23:07:00:fd:56*
The device address is used in *newDevice, getServices, bond, undbond, connect* and *disconnect.*

4.3.2 Services

Available services are:

Service	**Description**	**Implemented**
serial	Basic connection for sending SMS, reading phone book.	Yes
headset	Same as handsfree, but limited functions.	Yes
handsfree	Placing calls, route audio from device to gateway, etc.	Yes
avrcp	For controlling the media player.	**No**
obexftp	File transfer.	**No**
syncml	Synchronizing phone book and calendar events.	**No**
a2dp	Support streaming of high quality audio, like music.	**No**
dun	Dial-up internet. Used for internet connection.	Yes

Table 4.6: Services

Services are used in *getServices* and *getSupportedServices*.

4.3.3 Phone book storages

A phone book storage are a storage to read contacts from. Available phone book storages are listed in table 4.7.

Storage	**Description**
simFixDialingPhonebook	The number stored at the keypad buttons 0-9, usually called speed dialing.
simRecentCalls	Use this one to get the recent calls made by on the device. If this list is empty, try memoryRecentCalls.
memoryPhonebook	Standard phone book stored in mobile memory. If this list is empty, try simPhonebook.
simAndMemoryPhonebook	Combined phone book. Get all contacts from sim card and device memory.
simPhonebook	Standard phone book stored in mobile memory. If this list is empty, try memoryPhonebook.
memoryRecentCalls	Use this one to get the recent calls made by on the device. If this list is empty, try simRecentCalls.
simOrMemoryEmergencyNumber	Holds emergency numbers.
memoryUnansweredCalls	All unanswered/missed calls.
simOrMemoryOwnNumbers	The device mobile phone number.
memoryReceivedCalls	Received/answered calls.

Table 4.7: Phonebook storages

Phonebook storages are used in *getLists*, *getListParams* and *getList*.

4.3.4 List/storage entry type

A list entry/storage entry type is used to indicate what kind of number the number associated with the entry is. Available list entry type is listed in table 4.8.

Type	**Description**
unknown	The type is unknown or unavailable.
work	The number is a work phone number.
cell	The number is a cell phone number.
home	The number is a home phone number.
fax	The number is a fax phone number.
pref	Unknown, probably preferences.
pager	The number is a pager phone number.
msg	Unknown, probably a message.
other	The number is an other phone number.

Table 4.8: List entry types

4.3.5 Phone number

A phone number for placing a call or sending a SMS must be formatted using the international format. Numbers received from the gateway when reading from a phone book storage or when there is a new incoming call notification can be formatted either using the national or international format. In that case, it's up to the client to determine the number format. An example of an international number is: +46701234567

4.3.6 SMS storages

A SMS storage is the kind of memory to use to read from or write to. It is used in conjunction with the SMS storage type to set or get the memory to use for a certain storage type. Valid SMS storages are listed in table 4.9.

Storage	**Description**
device	Use device memory as memory source/destination.
sim	Use sim card as memory as memory source/destination.

Table 4.9: SMS storages

SMS storages are used in *getSMSStorages*, *getSMSList* and *readSMS*.

4.3.7 SMS storage types

A SMS storage type is used in conjunction with a SMS storage. The device has different memory settings for different types of actions. Valid SMS storages types are listed in table 4.10.

Storage type	**Description**
readDelete	Storage for reading and deleting SMS.
writeSend	Storage for writing and sending SMS.
receive	Storages where to receive SMS.

Table 4.10: SMS storage types

SMS storages are used in *getSMSStorages* and *setSMSStorages*.

4.3.8 SMS satus

A SMS status is a status of a SMS. Valid SMS statuses are listed in table 4.11.

Status	**Description**	**Implemented**
unread	Messages that are unread.	Yes
received	Messages that are received. This NOT include unread messages.	Yes
storedUnsent	Messages in outbox-folder, waiting to be sent.	**No**
storedSent	Messages in sent-folder.	**No**
all	All messages in all folders.	**No**
template	Template messages in the template folder.	**No**

Table 4.11: SMS statuses

SMS statuses are used in *getSMSList* and *readSMS*.

4.3.9 Encoding

Available encodings are are listed in figure 4.12.

Encoding	**Priority**
ISO-8859-1	First
UTF-8	Second
GSM	Third

Table 4.12: Encodings

The encoding applies to all data like contact names, SMS-messages, and so on **RECEIVED** from the gateway. After a successful serial, headset or handsfree connection, the client must make a call to getEncoding to determine the encoding in use, and then decode all SMS-text-body, contact names, etc. using that encoding. Data sent **TO** the gateway, like SMS-text-body, must be sent using the Western European (windows), Windows-1252 encoding. Note that the encoding cannot be set. The encoding is chosen in the priority order and to what encoding the device supports.

4.3.10 Battery status

Battery statuses is listed in table 4.13.

Status	Description
byBattery	The device is powered using battery.
hasBattery	The device has a battery, but is powered using another power source.
noBattery	The device does not have a battery.
powerFault	The device has very little power left, or another type of power error.

Table 4.13: Battery status

Battery status is used in *getBatteryStatus*.

4.3.11 Multiparty actions

Multiparty actions are used to handle three way dialing and conference calls, see table 4.14.

Action	Description
releaseAllHeldSetBusyWaiting	Releases all held calls or sets User Determined User Busy (UDUB) for a waiting call.
releaseAllActiveAcceptOther	Releases all active calls and accepts the other (waiting or held) call.
releaseActiveX	Releases the specific active call X.
holdAllAcceptOther	Places all active calls on hold and accepts the other (held or waiting) call.
holdAllExceptX	Places all active calls, except call X, on hold.
addHeld	Adds a held call to the conversation.
pairAndDisconnect	Connects two calls and disconnects the subscriber from both calls.

Table 4.14: Multiparty actions

"X" is the numbering (starting with 1) of the call given by the sequence of setting up or receiving the calls (active, held or waiting) as seen by the served subscriber. Calls hold their number until they are released. New calls take the lowest available number. Where both a held and a waiting call exists, the above procedures applies to the waiting call (that is, not to the held call) in conflicting situation.
Multiparty actions are used in *getSupportedMultiparty* and *doMultipartyAction*.

4.3.12 Events

An event is an incoming notification sent in the handsfree profile to indicate status changes. You shall use events to determine when calls are connected, held, etc. Not all devices have all events implemented. See table 4.15.

Name	**Description**
battchg	Battery charge level (0-5)
signal	Signal quality (0-5)
batterywarning	Battery warning (0-1)
chargerconnected	Charger connected (0-1)
service	Service availability (0-1) (Net contact status, 1 = Net contact)
sounder	Sounder activity (0-1) (Phone silent status, 1 = phone silent)
message	Message received (0-1)
call	Call in progress (0-1)
roam	Roaming indicator (0-1) (Home net status, 0 = Home Net)
smsfull	1: a short message memory storage in the MT has become full. 0: Memory locations are available
callsetup/call_setup	Bluetooth proprietary call set up status indicator. Possible values are as follows: 0: Not currently in call set up 1: Incoming call process ongoing 2: Outgoing call set up is ongoing 3: Remote party being alerted in an outgoing call
callheld	Indicator that indicates the status of any held calls on the AG. 0 = No held calls. 1 = Call on hold. If supported by ME.
vox	transmit activated by voice activity (0-1)

Table 4.15: Events

Events is used in event, *getAvailableEvents* and *getEventValue*.

4.3.13 Call status

Call status indicates the current status of all calls connected to the device. See table 4.16.

Status	**Description**
active	The call is active and ongoing.
held	The call is held.
dialing	The call is in the dialing state, not yet active.
alerting	The call is alerting the receiver about the call, i.e. playing the ring signal
. incoming	The call is a new incoming call.
waiting	The call is incoming waiting call.

Table 4.16: Call statuses

Call status are used in *getCallStatus*.

4.3.14 Audio for headset and handsfree

IMPORTANT! In some mobile phones, the gateway Bluetooth device needs to be added to the audio enabled Bluetooth device list, otherwise no audio will be routed from the device to the gateway. This is common in Sony-Ericsson devices.
There are two ways to use audio.

4.3.14.1 Using WaveIn and WaveOut

Audio is sent and received on the UDP-port pair associated with the service. The audio data is read and written using Windows audio API, waveIn and waveOut, and the client should use the same API. An UDP-packet contains the raw audio data recored by waveIn, found in the WAVEHDR->lpData. See appendix 4.3.14 at page 20 for an example class in C++.

WAVEFORMATEX variable	**Value**	**Description**
wFormatTag	WAVE_FORMAT_PCM	
nChannels	1	
nSamplesPerSec	8000	
nAvgBytesPerSec	16000	
nBlockAlign	1	
wBitsPerSample	16	
cbSize	2	
Non waveformatex variables	**Value**	**Description**
Block size	500	Size of each block sent on UDP.

Table 4.17: WaveIn/waveOut variables

4.3.14.2 Using btAudioClient.exe

btAudioClient.exe is an implementation of the source code found in appendix 4.3.14 at page 20. To use the application, launch it with the UDP audio in and UDP audio out data as the arguments. Example:
btAudioClient.exe 9103 9102
This example reads audio on port 9103, and sends audio on port 9102. **Note** that the application uses the audio device set as the default audio device in windows.

4.3.15 Error codes

Error codes:

Error	**Description**
errNoError	No error has occurred.
errNotConnected	Not connected to a device.
errAlredyConnected	Already connected to a device.
errUnableToConnected	Unable to connect to a device.
errCommandInProgress	A command is already in progress.
errCommandFailed	The command failed. Check for sub errors.
errWrongNumberOfArguments	Wrong number of arguments.
errInvalidArguments	One or more argument is malformed.
errCommandMalformed	The command is malformed.
errCommandAborted	Returned when a command is aborted.
errPortAlredyOpen	A connection or port is already open. Disconnect first.
errPortNotOpen	A port is not open.
errUnknown	Unknown error.
errNoCommandToAbort	Returned when trying to abort a command when there is no command running.
errAlredyBonded	The device is already bonded.
errNotBonded	Not bounded to the device.
errNoServices	No services, search for services for the device first.
errDeviceSearchInProgress	A device search is alredy in progress. Stop it first.
errNotSupported	Command is not supported.

Table 4.18: Error codes

4.3.16 Sub-error codes

Sub-error codes:

Sub-error	**Description**
subErrNoSubError	No sub-error.
subErrDeviceCannotBeFound	The device cannot be found. Search for services for the device first.
subErrServiceUnavailable	Search for services for the device first.
subErrTimeout	The command timed-out.
subErrUnknown	Unknown error.
subErrLineError	Rfcomm Line error. Disconnect and reconnect.
subErrPeerFailed	The device disconnected.
subErrNotWritten	The connection has failed, not all bytes of the command written.
subErrDeviceCommandError	The command is unsupported on the device.
subErrListDoesNotExists	The selected list does not exists.
subErrIndexIsOutOfRange	Index is out of range.
subErrBadPin	The pin is not valid.
subErrInvalidNumber	The phone number is not valid.
subErrMayStillPaired	Unbonding may have failed.
subErrConnectedOtherDevice	The service is connected to another device.
subErrNothingTodo	Nothing to do.

Table 4.19: Sub-error codes

Chapter 5

Finder commands- Device/Service search and bond/unbond

5.1 Device finding/handling commands

5.1.1 startDeviceSearch - Start searching for available devices

Starts a search for all devices nearby. When a new device is found, they are returned by the *newDevice* command. A device search stops when the *stopDeviceSearch* command are issued or when the search is done. Use command *getDeviceSearchStatus* to get current search status.

Syntax:
startDeviceSearch

Response:
startDeviceSearchReturn

Example:
Out: startDeviceSearch
Response: startDeviceSearchReturn

5.1.2 stopDeviceSearch - Stop searching for available devices

Stop searching for new devices.

Syntax:
stopDeviceSearch

Response:
stopDeviceSearchReturn

Example:
Out: stopDeviceSearch
Response: stopDeviceSearchReturn

5.1.3 getDeviceSearchStatus - Get status for device search

Get status for device search.

Syntax:
getDeviceSearchStatus

Response:
getDeviceSearchStatusReturn searching

Return arguments:
searching: Is 1 if searcing in progress, 0 otherwise.
Example:
Out: getDeviceSearchStatus
Response: getDeviceSearchStatusReturn 1
A device search is in progress.

5.1.4 getServices - Get services for a device

Get all available services for a device. Returns a list of all available services for a device. If there is no services available on the device, or if the device is not in range, an empty list is returned. Note that a name is not always returned using this function. Use *startDeviceSearch* to receive the device name.

Syntax:
getServices devAddr

Command arguments:
devAddr: A address to a device to search for services for.

Response:
getServicesReturn list *One device/service list.*

Return arguments:
list: A list of variable length. The first three elements are the same as in *newDevice*, see getDevices. Each next element in the list is a service name *(see section 4.3.2).* There could be zero or more additional elements (services).

Example:
Out: getServices 0:19:b7:7d:2:32
Response: getServicesReturn 00:19:b7:7d:02:32|Johboh-mobil|1|avrcp|headset|serial|syncml
This example requests for all services for the device 00:19:b7:7d:02:32, and get back a list of services which in this case is avrcp, headset, serial and syncml.

5.1.5 bond - Bond a device

Bond, or pair, the gateway with a device using a pin. The device (or the user of the device) must confirm with the same pin as the argument passed with the command.

Syntax: **bond devAddr{:}pin**

Command arguments:
devAddr: The address *(see section 4.3.1)* of the device to bond with.
pin: A alphanumeric string between one and 16 characters to use as authentication pin to pair the device.

Response:
bondReturn

Example:
Out: bond 00:19:b7:7d:02:32{:}123456
Response: bondReturn
This example tries to bond with the device 00:19:b7:7d:02:32 using pin 123456. The bondReturn indicates no error and a successful bond.

5.1.6 unbond - Unbond a device

Unbound, or unpair, a device from a gateway.

Syntax:
unbond devAddr

Command arguments:
devAddr: The address of the device to unbond.

Response:
unbondReturn

Example:
Out: unbond 00:19:b7:7d:02:32
Response: bondReturn

5.1.7 ping - Ping the gateway

Check if the gateway is alive and responding. If there is no response at all from the gateway within 10 seconds, the gateway can be considered dead.

Syntax:
ping

Response:
pingReturn

Example:
Out: ping

Response: pingReturn

5.1.8 getSupportedServices - Get the supported gateway services

Get the services that are implemented in the gateway.

Syntax:
getSupportedServices

Response:
getSupportedServicesReturn list *A list with zero or more elements.*

Return arguments:
list: A list of supported services.

Example:
Out: getSupportedServices
Response: getSupportedServicesReturn serial|headset
This gateway supports the serial and headset profile.

5.2 Incomming unsolicited commands

5.2.1 newDevice - New device found during device search

Sent from the gateway when a new device is found during a device search. Return information of a found device as an argument, where an argument contains the name and address of the device, and if the device is bonded or not. This command does not return any services for the device. See *getServices* for service discovery.

Syntax:
newDevice device *One device.*

Command arguments:
device: A list with three elements. First element is the device address *(see section 4.3.1)*, second argument is a human readable device name if the name are available, and the third element is 1 if the device is bounded, and 0 otherwise.

Example:
Response: newDevice 00:01:0a:6b:79:43:6d|VTECW464|0
A device with name VTECW464 that is unbonded.

Chapter 6

General commands for the serial, headset, handsfree, avrcp, obexftp, syncml, avrcp and a2dp service

6.1 Connection commands

6.1.1 connect - Connect to a device

Connect to a device using the service that is associated with the UDP-connection the command is sent on. The service needs to be available on the device, otherwise an error is returned. The gateway also needs to know about all available services for the device, therefore the client need to search for available services using *getServices* if this has not been done earlier during the lifespan of the gateway. If there is any error during connection, try to disconnect first, then connect again. If this connection is to serial, headset or handsfree service, use the getEncoding function to get the current encoding after a successful connection. Use *getStatus* to verify that the connection is successful.

Syntax:
connect devAddr

Command arguments:
devAddr: The address of the device to connect to.

Response:
connectReturn

Example:
Out: connect 00:19:b7:7d:02:32
Response: connectReturn
Connects to the device 00:19:b7:7d:02:32.

6.1.2 disconnect - Disconnect from a device

Disconnect a device using the service that is associated with the UDP-connection the command is sent on.

Syntax:
disconnect

Response:
disconnectReturn

Example:
Out: disconnect
Response: disconnectReturn

6.1.3 getStatus - Method name short description

Get the current connection status for the service that is associated with the UDP-connection the command is sent on.

Syntax:
getStatus

Response:
getStatusReturn status[**{:}devAddr**] *A status en an optional device address.*

Return arguments:
status: The current status. When status is connected, the gateway is connected to the service on a device with device address devAddr When status is disconnected, the service is not connected to any device.

Example:
Out: getStatus
Response: getStatusReturn connected{:}00:19:b7:7d:02:32

Chapter 7

Commands for serial, handsfree and headset service

7.1 Phone book specific commands

7.1.1 getLists - Get available phone book storages

Get all available phone book storages. This includes unanswered calls, recent calls list. See section 4.3.3 for list explanations.

Syntax:
getLists

Response:
getLists [**storage1**[**{:}storage2**[**{:}...**]]] *Zero or more storages.*

Return arguments:
storageX: A phone book storage.

Example:
Out: getLists
Response: getListsReturn memoryPhonebook{:}memoryRecentCalls{:}memoryUnansweredCalls

7.1.2 getListParams - Get parameters for a phone book storage

After getting all available phone book storages, and before fetching elements from a storage, a call to this function helps determine between which indexes there are valid entries. This function returns the start index and end index of where there are entries, as well as the maximum length of the name and number for a entry.

Syntax:
getListParams storage

Command arguments:
storage: One of the storages received when calling getLists.

Response:
getListParamsReturn list{:}startIndex{:}endIndex{:}maxNameLength{:}maxNumberLength

Return arguments:
list: The name of the list that parameters where requested for. startIndex: An integer. The list starts on this index.
endIndex: An integer. The list ends at this index.
maxNameLength: An integer. The max length for an entry name.
maxNumberLength: An integer. The max length for an entry number.

Example:
Out: getListsParams memoryRecentCalls
Response: getListParamsReturn memoryRecentCalls {:}1{:}20{:}50{:}50
This example requests for list parameters for the storage memoryRecentCalls. Valid indexes are between 1 and 20.

7.1.3 getList - Get all entries in a list/ from a storage

Get all entries between two indexes from a storage. A call to getLists and getListParams should precede this command. The name field is encoded using the encoding received from getEncoding. Note that not all indexes have entries, so if for example the start index is 1 and end index 500, not 500 entries will be returned unless all entries are non-empty.

Syntax:
getList storage{:}startIndex{:}endIndex

Command arguments:
storage: One of the storages received when calling getLists.
startIndex: A start index to start fetch from. Should be in range of the indexes received from getList-Params.
endIndex: The end index to stop fetching on. Should be in range of the indexes received from getList-Params.

Response:
getListReturn [**entry1**[**{:}entry2...**]] *Zero or more entries.*

Return arguments:
entryX: An entry is a list with four or five elements. The first element is the entry index, the second element is the type of the entry *(see section 4.3.4)*, the third is the name and the fourth is the number. If there is a date associated with the entry (available on some devices when reading from e.g. the memoryU-nansweredCalls storage), the list contains a fifth element with the date. The date format is not known, and should be parsed using a generic string to date parser. The usual format could be "yyyy/dd/mm hh:ii:ss" or "yyyy-mm-dd hh:ii:ss".

Example:

Out: getList memoryPhonebook {:}1{:}2
Response: getListReturn 1|work|Joe|01234567{:}2|home|John|+4274244212
This example get the first two entries in the memory phone book list.

7.2 Device functionality commands

7.2.1 getEncoding - Get the current encoding for received data

This function should be called after a successful connection to determine the encoding in use for data fields for received data like SMS-body-text, contact names etc.

Syntax:
getEncoding

Response:
getEncodingReturn encoding *One encoding.*

Return arguments:
encoding: The encoding in use. *(see section 4.3.9).*

Example:
Out: getEncoding
Response: getEncodingReturn ISO-8559-1

7.2.2 getBatteryStatus - Get battery level and status

Receives how the device is powered, and how much power there are left in the battery, if a battery is connected.

Syntax:
getBatteryStatus

Response:
getBatteryStatusReturn status{:}level

Return arguments:
status: The battery status *(see section 4.3.10).*
level: If 0, the battery is exhausted or no battery connected. If between 1 and 100, the level represent the percentage of capacity remaining.

Example:
Out: getBatteryStatus
Response: getBatteryReturn byBattery{:}78
The battery status indicates a capacity of 78% and that the device is powered by battery.

7.2.3 getSignalQuality - Get the signal quality

Receives the how strong the signal level is of the GSM or UMTS-network.

Syntax:
getSignalQuality

Response:
getSignalQualityReturn quality

Return arguments:
quality: The signal quality between 0-31 where 0 = -113 dBm or less, 1 = -111 dBm, 2..30 = -109... -53 dBm, 31 = -51 dBm or more.

Example:
Out: getSignalQuality
Response: getSignalQualityReturn 21

7.2.4 getValidRingVolume - Get valid ring volume interval

This function should be called before setRingVolume to know which value that is valid to use as an argument to setRingVolume.

Syntax:
getValidRingVolume

Response:
getValidRingVolumeReturn minValue{:}maxValue

Return arguments:
minValue: The minimum value.
maxValue: The maximum value.

Example:
Out: getValidRingVolume
Response: getValidRingVolumeReturn 0{:}8
The volume can be set to a value between 0 and 8.

7.2.5 setRingVolume - Set the ring volume

Set the ring volume to a value between the allowed one received by calling getValidRingVolume. The ring volume is the volume of the signal that is played on the device when there is a new incoming call.

Syntax:
setRingVolume volume

Command arguments:

volume: A value between the min and max received from getValidRingVolume to set the ring volume to.

Response:
setRingVolumeReturn

Example:
Out: setRingVolume 6
Response: setRingVolumeReturn
Sets the ring volume to 6.

7.2.6 getRingVolume - Get the current ring volume

Get the current ring volume.

Syntax:
getRingVolume

Response:
getRingVolume volume

Return arguments:
volume: The current ring value.

Example:
Out: getRingVolume
Response: getRingVolumeReturn 6
The ring volume is set to 6.

7.2.7 setSilence - Set silence mode

Set the phone in soundless mode or not. When set, the phone does not play the ring signal when there is a new incoming call. Depending on the device, no SMS signal will be played either.

Syntax:
setSilence enable

Command arguments:
enable: 1 to enable silence mode, 0 to disable it.

Response:
seSilenceReturn

Example:
Out: setSilence 1
Response: setSilenceReturn

Enable soundless mode.

7.2.8 getSilence - Get silence mode status

Get the status of the silence mode.

Syntax:
getSilence

Response:
getSilenceReturn enabled

Return arguments:
enabled: 1 when the soundless mode is enabled, 0 otherwise.

Example:
Out: getSilence
Response: getSilenceReturn 0
The silence mode is disabled.

7.2.9 setKeypressNotification - Enable or disable key press notification

Enable or disable incoming key press notification when the user press a key on the device.

Syntax:
setKeypressNotification enable

Command arguments:
enable: 1 to enable key press notification, 0 otherwise.

Response:
setKeypressNotificationReturn

Example:
Out: setKeypressNotification 1
Response: setKeypressNotificationReturn

7.3 Call specific commands

7.3.1 dial - Dial a number

Place a call to a number. The number must on the international format. Note that when using the handsfree service, you can make multiparty calls using dial together with doMultipartyAction.

Syntax:
dial number

Command arguments:
number: The number to dial.

Response:
dialReturn

Example:
Out: dial +701234567
Response: dialReturn

7.3.2 answer - Answer an incoming call

Answer an incoming call.

Syntax:
answer

Response:
answerReturn

Example:
Out: answer
Response: answerReturn

7.3.3 hangup - Hangup an ongoing or incoming call

Hangup an ongoing or reject an incoming call.

Syntax:
hangup

Response:
hangupReturn

Example:
Out: hangup
Response: hangupReturn

7.4 SMS specific commands

7.4.1 sendSMS - Create and send a new SMS

Send a new SMS to a receiver. The text must be encoded using Windows-1252 (standard windows encoding) and the number must be in the international format.

Syntax:
sendSMS number{:}text

Command arguments:
number: The receivers number on the international format.
text: The content of the SMS encoded in Windows-1252.

Response:
sendSMSReturn

Example:
Out: sendSMS +46701234567{:}Hello World!
Response: sendSMSReturn

7.4.2 getSMSStorages - Get SMS storages for a storage type

Get available SMS storages for a SMS storage type. Use this function to know valid storages to read SMS from. This function should be called before reading SMSes.

Syntax:
getSMSStorages type

Command arguments:
type: The SMS storage type to get storages for *(see section 4.3.7).*

Response:
getSMSStoragesReturn type{:}list *Zero or more elements with SMS storages.*

Return arguments:
type: The SMS storage type storages where requested for.
list: A list of available SMS storages.

Example:
Out: getSMSStorages readDelete
Response: getSMSStoragesReturn readDelete{:}device|sim
Available SMS storages for reading and deleting SMSes are device and sim memory.

7.4.3 setSMSStorage - Set the storage for a SMS storage type

This function is called inside readSMS and getSMSList to set the correct storage when reading an SMS. Use this function to set another memory for receiving SMS, if the device support it. Use getSMSStorages to check available storages.

Syntax:
setSMSStorage type{:}storage

Command arguments:
type: The SMS storage type to set SMS storage for.
storage: The SMS storage to set.

Response:
setSMSStorageReturn

Example:
Out: setSMSStorage receive{:}device
Response: setSMSStorageReturn
Set to use the device memory for storing received SMS.

7.4.4 readSMS - Read a SMS from a storage

Read a SMS from the readDelete storage type using the memory specified as an argument. The text field is encoded using the encoding received from getEncoding.

Syntax:
readSMS storage{:}index

Command arguments:
storage: The SMS storage to read from.
index: The index that specifies which SMS to read.

Response:
readSMSReturn sms *A SMS as a list.*

Return arguments:
sms: A SMS represented as a list. The first element is the SMS index. The second element is the SMS status. The third element is the sender number in national or international format. The fourth element is the received date on the format "yyyy-mm-dd HH:ii:ss". The fifth element is the SMS-body-text, encoded with the encoding received from getEncoding.

Example:
Out: readSMS device{:}123456
Response: readSMS 123456|received|+46701234567|2007-12-24 12:00:00|Hello

7.4.5 getSMSList - Get all SMSes in a storage

Receives all SMSes in a storage with a specific SMS status. Note that when reading an SMS with status unread, the status will change to received.

Syntax:
getSMSList storage{:}status

Command arguments:

storage: The storage to read from.
status: The function only returns SMSes with this SMS status.

Response:
getSMSListReturn [**sms1**[**{:}sms2**[**{:}...**]]] *Zero or more SMSes.*

Return arguments:
sms: A SMS represented as a list, same as in readSMS.

Example:
Out: getSMSList sim:unread
Response: 1|unread|+46112|2007-12-24 12:00:00|Hi!{:}2|unread|+46115|2007-12-25 12:00:00|Hello!
Fetch all SMSes from the sim card with status unread.

7.4.6 setSMSNotification - Enable or disable SMS notification

Enable or disable incoming SMS notification when receiving a new SMS. By default SMS notification is activated when using the handsfree service, but not when connecting to headset or serial.

Syntax:
setSMSNotification enable

Command arguments:
enable: 1 to enable SMS notification, 0 otherwise.

Response:
setSMSNotificationReturn

Example:
Out: setSMSNotification 1
Response: setSMSNotificationReturn

7.5 Incomming unsolicited commands

7.5.1 newSMS - New SMS received

When the device has received a new SMS, this command is sent containing the SMS storage the SMS has been stored in, and on which index. Use readSMS to read the received SMS.

Syntax:
newSMS storage{:}index

Command arguments:
storage: The storage where the SMS is stored.
index: The index where the SMS is stored.

Example:
Response: newSMS device{:}42672

7.5.2 ring - New incoming call

Sent periodically about every 3 seconds when there is an incoming call.

Syntax:
ring

Example:
Response: ring

7.5.3 number - Incoming number

On a new incoming call, this command reports the callers' number.

Syntax:
number nr

Command arguments:
nr: The callers number in national or international format.

Example:
Response: number +46702319191

7.5.4 keypress - A key has been pressed

This command is sent by the gateway when the user press a key on the device and key press notification is enabled.

Syntax:
keypress key{:}press

Command arguments:
key: The key that have been pressed or released.
press: 1 if the key has been pressed, or 0 if it was released.

Example:
Response: keypress 5{:}0
The key "5" was released.

Chapter 8

Commands for the headset, handsfree and a2dp service

8.1 Incomming unsolicited commands

8.1.1 audioEvent - Audio status changed

This event is sent from the gateway when audio is connected or disconnected. When connected, audio is sent and read on the respective audio UDP-connection pair for the current service.

Syntax:
audioEvent status

Command arguments:
status: connected when there exists an audio connection, and disconnected otherwise.

Example: Response: audioEvent connected

Chapter 9

Commands for the handsfree service

9.1 Call specific commands

9.1.1 getSupportedMultiparty - Get available call hold and three way actions

This command returns all supported three way calling/multiparty actions that the connected device supports. Note that the network provider may not support all available actions.

Syntax:
getSupportedMultiparty

Response:
getSupportedMultipartyReturn [**action1**[**{:}action2**[**{:}...**]]] *Zero or more actions.*

Return arguments:
actionX: A supported action *(see section 4.3.11).*

Example:
Out: getSupportedMultiparty
Response: getSupportedMultipartyReturn addHeld{:}releaseActiveX
The device support two multiparty actions, the "Add a held call to the conversation." and "Releases the specific active call X."

9.1.2 doMultipartyAction - Perform a multiparty action

Use getSupportedMultiparty to get all available multiparty actions, and use this command to perform one of them. See section 4.3.11.

Syntax:
doMultipartyAction action[**{:}X**]

Command arguments:
action: One of the actions from getSupportedMultiparty.
X: A call index. Needed for some of the multiparty actions.

Response:
doMultipartyActionReturn

Example:
Out: doMultipartyAction releaseActiveX{:}2
Response: doMultipartyActionReturn
Releases the active call number 2.

9.1.3 getCallStatus - Get status for all calls

Returns status for all active, held, incoming and waiting calls. Use this function to determine call index used in doMultipartyAction. If there are no calls, this function may fail on the device.

Syntax:
getCallStatus

Response:
getCallStatusReturn [**call1**[**{:}call2**[**{:}...**]]] *Zero or more calls.*

Return arguments:
callX: A call is a list with five elements. The first element is the call index used in doMutlipartyAction. The second element is 1 when the call is oriented into the device, and 0 when the call was made from the device. Element three is the status of the call *(see section 4.3.13).* The fourth element is 1 if this call is in a multiparty conference and 0 if it is not. The fifth and last element is the phone number this call is to/from.

Example:
Out: getCallStatus
Response: getCallStatusReturn 1|1|active|0|0701234567
Indicates that there is currently one active call from the number 0701234567 to the device.

9.1.4 sendDTMF - Send DTMF

Send DTMF tones during an ongoing call. Valid characters are 01234567890*#ABCD.

Syntax:
sendDTMF dtmf

Command arguments:
dtmf: A string of DTMF tones to send.

Response:
sendDTMFReturn

Example:
Out: sendDTMF *111#
Response: sendDTMFReturn

9.2 Device functionallity commands

9.2.1 getAvailableEvents - Get available events

Get the supported events by this device. See section 4.3.12.

Syntax:
getAvailableEvents

Response:
getAvailableEvents [**event1**[**{:}event2**[**{:}...**]]] *Zero or more events.*

Return arguments:
eventX: A event name.

Example:
Out: getAvailableEvents
Response: getAvailableEvents battchg{:}call{:}callsetup{:}sounder
This device support battery charger level, call, call setup and sounder events.

9.2.2 getEventValue - Get current value for an event

Get the value for one of the events received from getAvailableEvents.

Syntax:
getEventValue name

Command arguments:
name: The event name to get the value of.

Response:
getEventValueReturn name{:}value

Return arguments:
name: The event name.
value: The current value for the event.

Example:
Out: getEventValue call
Response: getEventValueReturn call{:}0
Indicates that the device is currently not in a call.

9.2.3 getNetworkOperator - Get the name of the network operator

Returns the network operators name.

Syntax:
getNetworkOperator

Response:
getNetworkOperatorReturn operator *A string.*

Return arguments:
operator: A string containing the name of the current selected network operator.

Example:
Out: getNetworkOperator
Response: getNetworkOperatorReturn Tele2Comviq

9.3 Incoming unsolicited commands

9.3.1 event - New event

Sent by the device on a new event. An example event can be when a call has been setup, when there is a new incoming call, when a call have been connected, when the charger is attached. See section 4.3.12. You should listen for events to know when calls are connected, ongoing etc.

Syntax:
event name{:}value

Command arguments:
name: The name of the event.
value: The new value for the event.

Example:
Response: event call{:}1
Indicates that a call is in progress.

9.3.2 callWaiting - New waiting incoming call

This command is sent during an ongoing call when there is a new incoming call.

Syntax:
callWaiting number

Command arguments:
number: The number of the incoming waiting call.

Example:
Response: callWaiting +46701234567

Chapter 10

Commands for the dun service

10.1 Connection commands

10.1.1 connect - Connect to a device

See first connect function in chapter 6 General commands for the serial, headset, handsfree, avrcp, obexftp, syncml, avrcp and a2dp service. Connects to a dial-up connection using the nomadic device as a modem. See Chapter *3 installation* first!

Syntax:
connect devAddr{:}phonenumber{:}username{:}password

Command arguments:
devAddr: The address of the device to connect to.
phonenumber: The number to dial.
username: Username.
password: Password.

Response:
connectReturn

Example:
Out: connect 00:19:b7:7d:02:32{:}*99#{:}{:}
Response: connectReturn
Create a dial-up connection using phone number *99# to device 00:19:b7:7d:02:32 using no password nor username.

10.1.2 disconnect - Disconnect from a device

See general *disconnect* function in chapter 6.1.

10.1.3 getStatus - Method name short description

See general *getStatus* function in chapter 6.1.

Chapter 11

Scenarios and usages

This chapter contains a couple of user scenarios and examples.

11.1 Parsing and creating a command

See appendix 14.2 respectively 14.3 for a C# and C++ class for parsing and creating a command.

11.2 Find and automatically connect to a device

This section describes the procedure to find a device, and then automatically discover and connect to it when it is in range. GUI is a user interface that the driver of the car have access to. The user is the driver of the car who uses the GUI.

One time bonding:

1. send *startDeviceSearch* and wait for the *newDevice* until *getDeviceSearchStatus* return 0 or until a specific device is found. Display the device list in the GUI.
2. Let the user choice a device he/she wants to bond with.
3. send *bond <deviceaddr>{:}<pin>* to bond with the device, where *<deviceaddr>* is the device address received in step 1. *<pin>* is a random chosen pin displayed in the GUI.
4. The user enters the pin displayed in the GUI in his device to accept the bonding.
5. Wait for *bondReturn*.
6. Store the *<deviceaddr>* in the GUI in a list of accepted devices to automatically connect to.
7. Use *getServices <deviceaddr>* to receive services for the device and display for the user.
8. Let the user chose the services he/she want to connect to.
9. Store the chosen services in the GUI in a list of services to connect to.

To automatically connect:

1. Periodically use *getServices <deviceaddr>* using the device address stored in the "automatic connect" device list stored in the GUI.

2. When services are received that match services in the auto connect list, go to step 3. Else go to step 1.
3. Connect to the service by sending *connect <deviceaddr>* on the UDP-connection for the current service.
4. Wait for *connectReturn*. In case of an error, send *disconnect* and then go to step 3 and try to connect again.
5. After receiving a *connectReturn*, send *getStatus* to verify that device is connected.
6. At this point, the device is connected.
7. You should use *getEncoding* to get the current encoding in use to decode SMS-body-text and phone book entries.

11.3 Reading phone book entries

This section explains how to read entries from the phone book.

This scenario reads all entries from the "missed calls"-list, which is named *memoryUnansweredCalls*. Note that we assume that this list exists. To verify, use *getLists* to get all available lists.

1. First we need to know which indexes there exists entries in the list. Send *getListParams memoryUnansweredCalls* the get list parameters.
2. Wait for *getListParamsReturn memoryUnansweredCalls{:}<startIndex>{:}<endIndex>{:}<a>{:}<b>* where *<startIndex>* and *<endIndex>* are the only arguments we are interested in this example. Start index and end index are the indexes where the list starts and ends.
3. Now we want to get all entries. Send *getList memoryUnansweredCalls{:}<startIndex>{:}<endIndex>*, where the *<startIndex>* and *<endIndex>* are values received from step 2.
4. Wait for *getListReturn <entry1>:<entry2>....* And parse each entry accordingly. See *getList* for more information.

11.4 Reading SMSes

This section explains how to read SMSes.

In this example we want to read all unread SMSes stored in the device-memory. Note that we assume that the device memory are available. To verify, use *getSMSStorages readDelete* to get all available memory storages for the *readDelete* storage, which is the only one we can read from.

1. Send *getSMSList device{:}unread* to read all unread SMSes from the device memory. Note that all SMSes will not longer have the status set to unread after this command, because we are reading them. They will instead have the status set to received.
2. Wait for *GetSMSListReturn sms1{:}sms2...* and parse the SMS accordingly. See *getSMSList* for more information.

11.5 Call handling for the handfree service

This section covers some examples for placing regular and third party calls as well as reading/receiving call and phone statuses.

11.5.1 Placing a call

This is a simple example for placing a call, knowing when the remote party has answered and hanging up.

1. Placing the call, send *dial <number>* where *<number>* is the number you want to dial, in international format, for example *dial +47702319191*
2. Wait for *dialReturn*.
3. To know what's happening with the call and when the remote party has answered, you need to listen for incoming events, especially the *callsetup* and *call* events. When the call is initiating, the *callsetup* is set to value 2 indicating that "Outgoing call set up is ongoing". When the remote device is ringing, *callsetup* changes to value 3. When the remote device answers the call, *callsetup* changes it value to 0, because the device is no longer in a call setup, as the same time as the *call* event is set to 1, indicating an ongoing call.
4. When receiving the *audioEvent <status>* command, check the status for connected and start reading and sending audio to the UDP-audio pair for the handsfree audio.
5. To hang up, send *hangup*, wait for *hangupReturn* and the event *call* with value 0 indicating that the call has ended.

11.5.2 Receiving a call

Receiving and answering an incoming call.

1. Wait for an incoming *ring* or *number <nr>* command or a *callsetup* event with value 1. This entire three indicates an incoming call. Play your own ring signal every *ring* command, or the in-band-ring-tone if available, and notify the user of who is ringing by displaying the *<nr>* number from the *number* command.
2. To answer, send *answer* and wait for *answerReturn*, a *callsetup* event of value 0 and a *call* event of value 1.

11.5.3 Multiparty calls

In both examples below, we assume that multiparty calls are supported by the device and the network. Use the *getSupportedMutliparty* command to check the device support.

This first example shows how to place two outgoing calls and connect them to a conference call, and then disconnect the first call from the conversation.

1. Establish the first call as in Scenario 11.5.1.
2. Send *doMultipartyAction holdAllAcceptOther* to hold the ongoing call.
3. Establish the second call the same way as in step 1.

4. To add the first held call to the conversation, send *doMultipartyAction addHeld*.

5. You are now in a conference call with two parties.

6. The calls receive identification numbers in the order they are established. In this case the first call has index 1 and the second index 2. We can check the status and indexes of the call by sending *getCallStatus*.

7. To release the first call from the conversation, send *doMultipartyAction releaseActiveX{:}1*, where the second argument is the index of the call to release.

8. Use *getCallStatus* to verify that the call is no longer active.

This second example shows how to make a call and receive a new call and switch between them.

1. Establish a call as in Scenario 11.5.1.

2. On a new incoming call during an ongoing call, the *callWating <number>* is sent from the gateway. Use the *<number>* to notify the user of the incoming call.

3. To accept the incoming call and set the current call on hold, send *doMultipartyAction holdAllAcceptOther*. Use *getCallStatus* to check verify the status of the calls.

4. To toggle between the calls, send *doMultipartyAction holdAllAcceptOther*.

Chapter 12

Troubleshooting

12.1 Problems and solutions

12.1.1 How to update a bug in this manual?

This manual is written in LaTeX. The LaTex IDE TeXnicCenter 1 Beta 7.01 together with MikTex 2.7 where used during development. Figures where created in Microsoft Visio 2007.
MikTex (install first): `http://miktex.org`
TeXnicCenter: `http://www.toolscenter.org/`

12.1.2 Error messages

12.1.2.1 Invalid argument device. The device cannot be found.

The gateway does not know about the device yet. Search for devices first, or search for services for a device address.

12.1.2.2 Another audio connection is active to the requested remote device.

There already exists an audio connection to the device you are trying to connect to. This occurs when you already have an open device connection, either with NDGate or using native Windows Bluetooth functions, or if you ended a previous connection in the wrong way. Due to the Bluetooth stack instability issues, the only solution to this is to reboot the computer.

12.1.2.3 An unknown or internal error occured./Unknown error.

This can happen at any time, and is Bluetooth stack related. Try these steps, and stop when the problem has been solved.

1. Disconnect devices from NDGate by using the disconnect command.
2. Restart Bluetooth stack.
3. Restart nomadic device.
4. Restart NDGate.
5. Restart Computer.

12.1.2.4 Runtime errors

You may receive runtime errors. These errors should not happen at all.

- "Semaphore creation failed." - Semaphore creation in Audio class failed.
- "WaveIn or/and WaveOut thread creation failed./Error opening wave playback/input device." - Audio class waveIn/Out failure. Check that you have Bluetooth audio installed and that your audio device have recent drivers.
- "Unable to find bluetooth-device!" - You dont have an bluetooth audio device installed. Reinstall Bluetooth dongle drivers.

There are more less common runtime errors. When they occur, search for the error message in the source code to find where it occurred and try to recreate the error.

12.1.2.5 No UDP data is received when using infoTainer.

Restart infoTainer.

12.1.2.6 Audio routing does not work.

1. Establish an voice call and wait up to 2 minutes before you
2. Check that you have a Bluetooth audio device installed. If not, reinstall Bluetooth dongle drivers.
3. You must set a non-Bluetooth audio device as your primary audio device in windows control panel -> "Sound and audio devices"
4. If using infoTainer, make sure that the btAudioClient is launched.
5. Make sure that the device you are connecting to have granted Bluetooth audio access for the gateway PC. Check your Bluetooth settings on your device.
6. Try to restart NDGate, your device and the computer.

Chapter 13

Wordlist

- UDP - http://en.wikipedia.org/wiki/User_Datagram_Protocol
- API - http://en.wikipedia.org/wiki/API
- GUI - http://en.wikipedia.org/wiki/GUI

Chapter 14

Appendix

14.1 Handsfree workaround

Due to a bug in the Bluetooth stack, the handsfree audio gateway is not always found by NDGate. It possible to manually add the handsfree service port number as an configuration settings to NDGate. To do that, you need to know the SCN-port number for the handsfree gateway on each device you want to enable. Follow this guide to discover the SCN-port number.

You will need to have the application spylite.exe and btHandsfreeFinder.exe.

Step one is to configure spylite.exe.

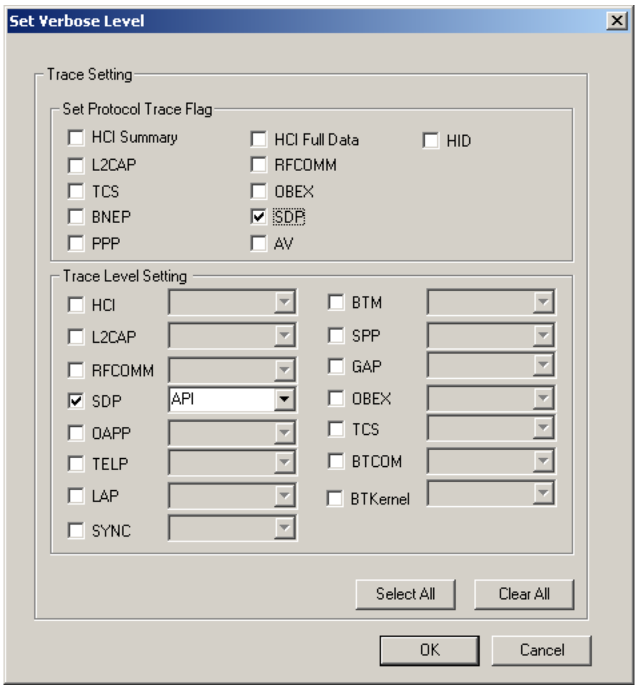
1. Start spylite.exe.

2. If you get a pop-up saying that you need to logout and login again, close the popup and restart your computer.

3. After restart, or if you did not get an pop-up, go to menu ”Tools” -> ”Set protocol trace..”

4. Uncheck all items execpt SDP, and select API under ”Trace level settings”. It should look like in figure 14.1.

5. Leave spylite open.


Set Verbose Level

Trace Setting

Set Protocol Trace Flag

☐ HCI Summary	☐ HCI Full Data	☐ HID
☐ L2CAP	☐ RFCOMM	
☐ TCS	☐ OBEX	
☐ BNEP	☑ SDP	
☐ PPP	☐ AV	

Trace Level Setting

☐ HCI		☐ BTM	
☐ L2CAP		☐ SPP	
☐ RFCOMM		☐ GAP	
☑ SDP	API	☐ OBEX	
☐ OAPP		☐ TCS	
☐ TELP		☐ BTCOM	
☐ LAP		☐ BTKernel	
☐ SYNC			

Figure 14.1: Spylite configuration

Step two is to search for handsfree service.

1. Enable Bluetooth on the device you want to add the handsfree for. Also make sure it discoverable.
2. Open btHandsfreeFinder.exe
3. Follow the guide in the window, the same guide as follows here.
4. Disable your Bluetooth stack. Do this by right-clicking on the Bluetooth icon in the taskbar en select "stop" or "disable", or goto the "Windows start menu"->"Settings"->"Control panel"->"System"->"Hardware tab"->"Device manager", find the Bluetooth device, right-click and disable it.
5. Enable the Bluetooth stack again.
6. Start a device search and wait until the search is complete. You will get a message about that.
7. Select the device you want to search for handsfree on.
8. Goto the spylite.exe application window, click the "Erase" button and then press enter in the btHandsfreeFinder window.
9. Let the application search for services.
10. When done, click the "Scroll" button in spylite to disable new log-messages.

Step three is to find out the SCN-port number from the log.

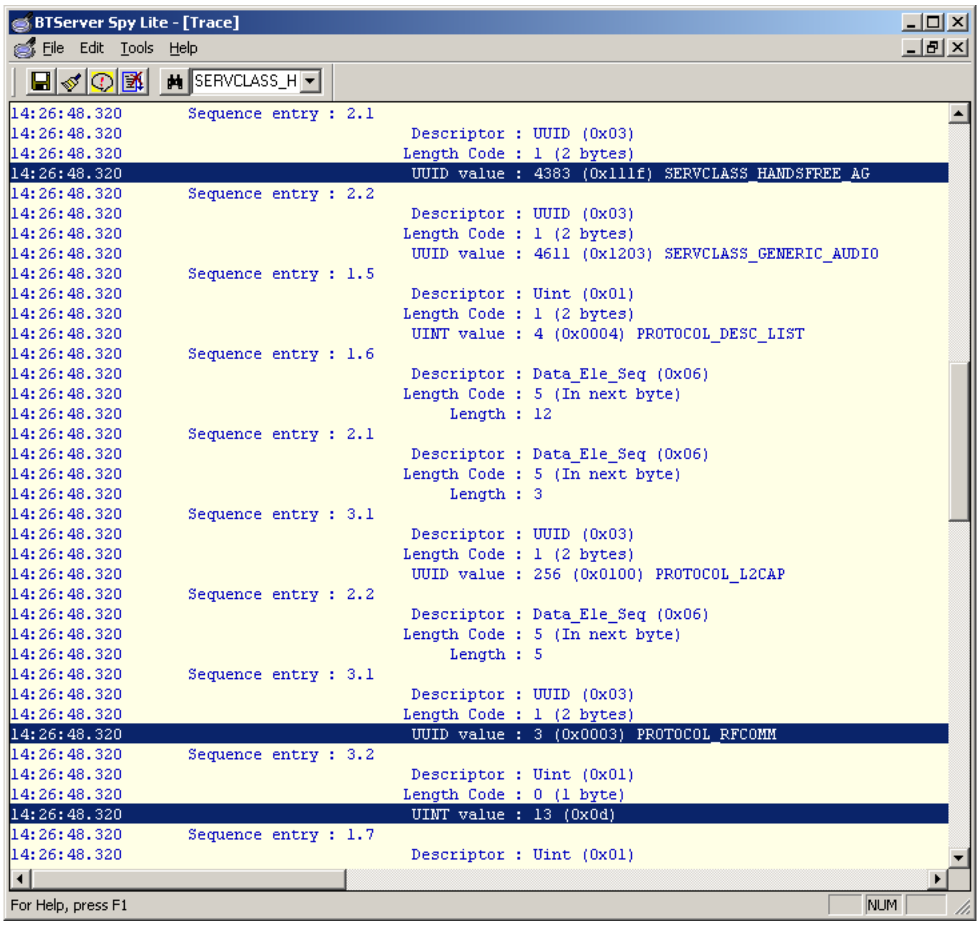
1. Goto menu "Edit"->"Find in trace". type *SERVCLASS_HANDSFREE_AG* in the box and click "find all".
2. If you don't find any, the device has no handsfree audio gateway. If you are sure it does, goto to the top of this section and try again.

3. In the same are around where you found the *SERVCLASS_HANDSFREE_AG*, you will see a couple of "Sequence entry" entries, in blue color. Manually look and find the text "PROTOCOL_RFCOMM" as the UUID value, in the same blue text/yellow background area in one of the sequence entries. When found, the handsfree SCN-port number is the UINT value of the very next sequence entry. See figure 14.2.

Step four is to configure NDGate.
Open the settings.ini that is in the same directory as the NDGate.exe application. Set the fields *handsfreeDevices* and *handsfreeScns* with the device/scn you want to add, separate each device and SCN you with comma. Eg, if you want to add two devices, 00:19:B7:7D:02:32 with SCN 13 and 00:1d:28:76:9b:ab with SCN 9, it should look like this:

handsfreeDevices=00:19:b7:7d:02:32,00:1d:28:76:9b:ab
handsfreeScns=13,9

For the handsfree service in figure 14.2, the SCN-port number is 13.



```
14:26:48.320      Sequence entry : 2.1
14:26:48.320                             Descriptor : UUID (0x03)
14:26:48.320                            Length Code : 1 (2 bytes)
14:26:48.320                             UUID value : 4383 (0x111f) SERVCLASS_HANDSFREE_AG
14:26:48.320      Sequence entry : 2.2
14:26:48.320                             Descriptor : UUID (0x03)
14:26:48.320                            Length Code : 1 (2 bytes)
14:26:48.320                             UUID value : 4611 (0x1203) SERVCLASS_GENERIC_AUDIO
14:26:48.320      Sequence entry : 1.5
14:26:48.320                             Descriptor : Uint (0x01)
14:26:48.320                            Length Code : 1 (2 bytes)
14:26:48.320                             UINT value : 4 (0x0004) PROTOCOL_DESC_LIST
14:26:48.320      Sequence entry : 1.6
14:26:48.320                             Descriptor : Data_Ele_Seq (0x06)
14:26:48.320                            Length Code : 5 (In next byte)
14:26:48.320                                 Length : 12
14:26:48.320      Sequence entry : 2.1
14:26:48.320                             Descriptor : Data_Ele_Seq (0x06)
14:26:48.320                            Length Code : 5 (In next byte)
14:26:48.320                                 Length : 3
14:26:48.320      Sequence entry : 3.1
14:26:48.320                             Descriptor : UUID (0x03)
14:26:48.320                            Length Code : 1 (2 bytes)
14:26:48.320                             UUID value : 256 (0x0100) PROTOCOL_L2CAP
14:26:48.320      Sequence entry : 2.2
14:26:48.320                             Descriptor : Data_Ele_Seq (0x06)
14:26:48.320                            Length Code : 5 (In next byte)
14:26:48.320                                 Length : 5
14:26:48.320      Sequence entry : 3.1
14:26:48.320                             Descriptor : UUID (0x03)
14:26:48.320                            Length Code : 1 (2 bytes)
14:26:48.320                             UUID value : 3 (0x0003) PROTOCOL_RFCOMM
14:26:48.320      Sequence entry : 3.2
14:26:48.320                             Descriptor : Uint (0x01)
14:26:48.320                            Length Code : 0 (1 byte)
14:26:48.320                             UINT value : 13 (0x0d)
14:26:48.320      Sequence entry : 1.7
14:26:48.320                             Descriptor : Uint (0x01)
```

Figure 14.2: Handsfree service spylite sequence entries

14.2 C# command parsing class

14.2.1 File btCommand.cs

```
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace infoTainer
{
    class btCommand
    {
        private string func;
        private List<string> arguments = new List<string>();
        private bool bIsValid;

        ///////////////////////  COMMAND PARSING ///////////////////
        // Syntax:
        // command param1{:}param2{:}param3\r\n         -> command and three params
        // command param1\r\n                                           -> command and one param
        // command param1{:}{:}param3\r\n                       -> command and three params, param 2 empty
        // command\r\n                                                          -> command, no params
        // command \r\n                                                         -> command and one empty param
        // command {:}\r\n                                                      -> command and two empty params
        // command {:}{:}\r\n                                                   -> command and three empty params
        //  command\r\n                                                         -> INVALID
        // \r\n                                                                         -> INVALID

        public btCommand(string rawData)
        {
            // First check that the rawData string is valid

            // Find \r\n which is the end. Search from begining.
            // If not found, invalid command.
            int end = rawData.IndexOf("\r\n", 0);
            if (end < 0) { bIsValid = false; return; }

            // Check that the first character starts with a non space
            // If so, the string is considered valid.
            if (end < 1 || Char.IsWhiteSpace(rawData[0])) { bIsValid = false; return; }

            // The command is valid.
            bIsValid = true;


            // Get function.
            // function is between start and space or end, whichever comes first.
            int space = rawData.IndexOf(" ", 0);

            if (space < 0) space = end;

            func = substring(rawData, 0, Math.Min(space, end) - 1);

            // Get arguments

            // If the argument index is 0, the argument value is between
            // space and either the next sep or end.
            // else if the argument is n>0, the argument is between n-1 sep
            // and the next sep or end.

            int start = 0;

            for (int argIndex = 0; true; argIndex++)
            {
                // First, special case, find start position.
                if (argIndex == 0) start = rawData.IndexOf(" ", 0);
                else
                {
                    start = rawData.IndexOf("{:}", start);
                    // add how much longer the sep are from a single char space.
                    if (start > -1) start += 2;
                }

                // Something found? No? Break then.
                if (start < 0) break;

                start++;

                // Next, find next sep or end.
                end = rawData.IndexOf("{:}", start);
                if (end < 0) end = rawData.IndexOf("\r\n", start);

                // End found, substring.
                arguments.Add(substring(rawData, start, end - 1));

            }

            // Done.
        }
        public btCommand(string function, List<string> arguments)
        {
            func = function;
            this.arguments = arguments;
        }

        public string getFunction()
        {
            return func;
        }
        public string getArgument(uint index)
        {
            if (index >= arguments.Count) return "";
            return arguments[(int)index];
        }
        public List<string> getArguments()
        {
            return arguments;
        }
        public int numArguments()
        {
            return arguments.Count;
        }
        public bool isValid()
        {
            return bIsValid;
        }

        public string getCommandString()
        {
            string rstring = "";
            rstring += func;
            if (arguments.Count > 0)
            {
                rstring += " ";
                // Special case, add first argument
                rstring += arguments[0];
                // For each other arguments
                for (uint i = 1; i < arguments.Count; i++)
                {
                    rstring += "{:}";
                    rstring += arguments[(int)i];
                }
            }

            // Add end
            rstring += "\r\n";

            return rstring;
        }

        private string substring(string str, int startIndex, int endIndex)
        {
            return str.Substring(startIndex, endIndex - startIndex + 1);
        }

    }
}
```

14.3 C++ command parsing class

14.3.1 File btCommand.h

```
#include "stdafx.h"
#include "functions.h"

#pragma once

// Class for creating and parsing a Command received on UDP.

class CbtCommand
{
public:
        CbtCommand();
        // Create a new command from a raw string
        CbtCommand(string rawData);
        // Create a new command from a function and a list of arguments.
        CbtCommand(string function, vector<string> *arguments);
        ~CbtCommand();
public:
        // Command handle functions
        string getFunction(void);
        string getArgument(unsigned int index);
        vector<string> getArguments(void);
        int numArguments(void);
        bool isValid();

        string getCommandString(void);

        void CommandDEBUGChecker(void);

private:
        // Variables
        string func;
        vector<string> arguments;
        bool bIsValid;



};
```

14.3.2 File btCommand.cpp

```
#include "stdafx.h"
#include "btCommand.h"


//////////////////////  COMMAND PARSING ////////////////////
// Syntax:
// command param1{:}param2{:}param3\r\n         -> command and three params
// command param1\r\n                                           -> command and one param
// command param1{:}{:}param3\r\n                       -> command and three params, param 2 empty
// command\r\n                                                          -> command, no params
// command \r\n                                                         -> command and one empty param
// command {:}\r\n                                                      -> command and two empty params
// command {:}{:}\r\n                                           -> command and three empty params
//  command\r\n                                                         -> INVALID
// \r\n                                                                         -> INVALID



// Constructor, deconstructor

/*
* Create an empty command.
*/
CbtCommand::CbtCommand()
{
        this->bIsValid=false;
}

/*
* Create a new command from a raw data string.
*       param rawData the string to parse.
*/
CbtCommand::CbtCommand(string rawData)
{
        // First check that the rawData string is valid

        // Find \r\n which is the end. Search from begining.
        // If not found, invalid command.
        string::size_type end = rawData.find("\r\n",0);
        if (end == string::npos) { bIsValid = false; return; }

        // Check that the first character starts with a non space
        // If so, the string is considered valid.
        if (end<1 || isspace(rawData.at(0))) { bIsValid = false; return; }

        // The command is valid.
        bIsValid = true;


        // Get function.
        // function is between start and space or end, whichever comes first.
        string::size_type space = rawData.find(" ",0);

        func =  substring(rawData,0,min(space,end)-1);

        // Get arguments

        // If the argument index is 0, the argument value is between
        // space and either the next sep or end.
        // else if the argument is n>0, the argument is between n-1 sep
        // and the next sep or end.

        string::size_type start = 0;

        //01234567891011
        // 1{:}{:}{:}\r\n

        for(int argIndex=0;true;argIndex++)
        {
                // First, special case, find start position.
                if (argIndex == 0) start = rawData.find(" ",0);
                else
                {
                        start = rawData.find("{:}",start);
                        // add how much longer the sep are from a single char space.
                        if (start!=string::npos) start += 2;
                }

                // Something found? No? Break then.
                if (start==string::npos) break;

                start++;

                // Next, find next sep or end.
                string::size_type end = rawData.find("{:}",start);
                if (end == string::npos) end = rawData.find("\r\n",start);

                // End found, substring.
                arguments.push_back (substring(rawData,start,end-1));

        }

        // Done.
```

```
}

/*
* Create a new command from a supplied function and a argument list.
*       param function the function.
*       param inArguments the argument list.
*/
CbtCommand::CbtCommand(string function, vector<string> *inArguments)
{
        // TODO DEBUG check functions for spaces

        this->func = function;
        this->arguments.assign(inArguments->begin(),inArguments->end());
        this->bIsValid = true;
}

CbtCommand::~CbtCommand()
{

}

// public functions
/*
* Get the function of the command.
*       return the function of the command if the command is valid.
*/
string CbtCommand::getFunction()
{
        return func;
}

/**
* Get the argument at position index.
*       param index the param to get. Index start at 0.
*       return the argument.
*/
string CbtCommand::getArgument(unsigned int index)
{
        if (index>=arguments.size()) return "";
        return arguments.at(index);
}

/**
* Get the arguments.
*       return the arguments.
*/
vector<string> CbtCommand::getArguments(void)
{
        return arguments;
}


/*
* Get number of arguments.
*       return the number of arguments.
*/
int CbtCommand::numArguments()
{
        return (int)arguments.size();
}

/*
* return if the argument is valid or not.
*/
bool CbtCommand::isValid()
{
        return bIsValid;
}

// public static functions

/*
* Get the string for this command.
*       return the string for this command.
*/
string CbtCommand::getCommandString()
{
        string rstring = "";
        rstring.append(this->func);
        if (this->arguments.size()>0)
        {
                rstring.append(" ");
                // Special case, add first argument
                rstring.append(this->arguments.at(0));
                // For each other arguments
                for (unsigned int i=1;i<this->arguments.size();i++)
                {
                        rstring.append("{:}");
                        rstring.append(this->arguments.at(i));
                }
        }

        // Add end
        rstring.append("\r\n");
        return rstring;
}
```

14.4 C++ audio streaming

14.4.1 File audio.h

```
#include "stdafx.h"
#include "udp_shared.h"
#include "udpreceiver.h"
#include "udpsender.h"
#include "Mmsystem.h"
#include <mmreg.h>
#include "Thread.h"
#include "windows.h"
#include <vector>
#include <queue>
#include <list>

using namespace std;

#pragma once

#define SAMPLE_RATE 8000
#define BITS_PER_SAMPLE 16
#define CHANNELS 1
#define AVG_BYTES_PER_SEC ((BITS_PER_SAMPLE/8)*SAMPLE_RATE*CHANNELS)
#define BLOCK_SIZE (AVG_BYTES_PER_SEC/32)
#define NUM_BLOCKS (AVG_BYTES_PER_SEC/BLOCK_SIZE)

#define MAX_DELAY_MSEC  300
#define UPPER_THRESHOLD 3 // Delay threshold
#define LOWER_THRESHOLD 0 // Delay threshold
#define MAX_WOUT_BUFFERS (MAX_DELAY_MSEC/(((double)BLOCK_SIZE/(double)AVG_BYTES_PER_SEC)*1000.0))



class Caudio
{
public:
        enum AudioService{asHandsfree,asA2DP,asHeadset};
        enum ErrEnums {eeUdpInInUse, eeUdpOutInUse,eeNotInitilized,eeFailedOpenDevice,eeCriticalError,eeBluetoothDeviceNotFound };

private:
        bool getBluetoothDeviceId(UINT *idIn, UINT *idOut);
        static DWORD WINAPI WaveInFullThread(LPVOID);
        static DWORD WINAPI WaveOutEmptyThread(LPVOID);

public:
        Caudio();
        ~Caudio();
        bool start(void);
        bool initialize(int udpIn,int udpOut,AudioService as);
        bool stop(void);
        void getLastError(ErrEnums *err, string *errStr);
        void udpReceiveCallback(char * data, int size);

public:

struct WinBuffer
{
        WAVEHDR m_WaveHeader; // wave header for the buffer
        BYTE m_Data[BLOCK_SIZE];

        MMRESULT Prepare(HWAVEIN hWaveIn)
        { // Prepare for playback
                ZeroMemory(&m_WaveHeader, sizeof(m_WaveHeader));
                m_WaveHeader.dwBufferLength = BLOCK_SIZE;
                m_WaveHeader.lpData = (char*)(m_Data);
                m_WaveHeader.dwUser = (DWORD)this;

                return waveInPrepareHeader(hWaveIn, &m_WaveHeader, sizeof(m_WaveHeader));
        }

        MMRESULT Unprepare(HWAVEIN hWaveIn)
        {
                return waveInUnprepareHeader(hWaveIn, &m_WaveHeader, sizeof(m_WaveHeader));
        }

        MMRESULT Add(HWAVEIN hWaveIn)
        {
                return waveInAddBuffer(hWaveIn, &m_WaveHeader,sizeof(m_WaveHeader));
        }
};

struct WoutBuffer
{
        WAVEHDR m_WaveHeader;
        BYTE m_Data[BLOCK_SIZE];
        MMRESULT Prepare(HWAVEOUT hWaveOut)
        {
                ZeroMemory(&m_WaveHeader, sizeof(m_WaveHeader));
                m_WaveHeader.dwBufferLength = BLOCK_SIZE;
                m_WaveHeader.lpData = (char*)(m_Data);
                m_WaveHeader.dwUser = (DWORD)this;

                return waveOutPrepareHeader(hWaveOut, &m_WaveHeader, sizeof(m_WaveHeader));
        }

        MMRESULT Unprepare(HWAVEOUT hWaveOut)
        {
                return waveOutUnprepareHeader(hWaveOut, &m_WaveHeader, sizeof(m_WaveHeader));
        }

        MMRESULT Add(HWAVEOUT hWaveOut)
        {
                return waveOutWrite(hWaveOut, &m_WaveHeader,sizeof(m_WaveHeader));
        }
};

struct GlobalVariables
{
        queue<WoutBuffer*> playbackList;
        queue<WoutBuffer*> freeOutList;

        UdpSender * udpsender;
        Thread * udpThreadRec;

        WinBuffer inBlocks[NUM_BLOCKS];
        WoutBuffer outBlocks[NUM_BLOCKS*2];

        HWAVEOUT hWaveOut;
        HWAVEIN hWaveIn;

        int numWaveOutBuffers;
        int numWaveInBuffers;

        bool delay;
        bool closing;
        bool initilized;
        bool started;
};

private:
GlobalVariables globalvars;
string lastErrStr;
ErrEnums lastErr;

HANDLE semGlobal;

HANDLE WaveInFullThreadHandle;
HANDLE WaveOutEmptyThreadHandle;

DWORD WaveInFullThreadID;
DWORD WaveOutEmptyThreadID;

public:
        static void managePlaylist(GlobalVariables *global);
        static string Merr(MMRESULT m);
        static string getSocketErrorString(int error);
        static string Report(const char * str)
        {
                #ifndef DONT_USE_BLUETOOTH
                        Clogging::logg(str);
                #else
                        printf("%s\n",str);
                #endif

                return string(str);
        }

};
```

14.4.2 File audio.cpp

```
#include "stdafx.h"
#include "audio.h"

string semAudio;

// Create udp receiver object
template <class T>
class UDPReceiverObject : public IRunnable
{
        public:
                UdpReceiver *udpRec;
                char incommingBuff[UDP_PACK_MAXSIZE];
                int lastErrnr;

                UDPReceiverObject(int portnr,T *who,void (T::*func)(char * buff, int size)):callee(who),callback(func)
                {
                        // Create new udp receiver object.
                        lastErrnr = 0;
                        udpRec = udpReceiverCreate(portnr,UDPBLOCK);
                        if (udpRec==0) lastErrnr=getRecError();
                        _continue = true;
                }

                int getError() { return lastErrnr; }

                virtual unsigned long run()
                {
                        while(_continue)
                        {
                                // run this thread procedure
                                // Read udp data.
                                int byteReceived = udpReceiverReceive(udpRec,incommingBuff,UDP_PACK_MAXSIZE);
                                (callee->*callback)(incommingBuff,byteReceived);
                        }
                        return 0;
                }

                virtual void stop()
                {
                        _continue = false;
                        udpReceiverClose();
                }

        private:
                T *callee;
        void (T::*callback)(char * buff, int size);

        protected:
                bool _continue;
};

UDPReceiverObject<Caudio> * udprec;

//      Constructors and deconstructors
Caudio::Caudio()
{
        globalvars.initilized = false;
        globalvars.started = false;
}

bool Caudio::initialize(int udpIn, int udpOut, AudioService as)
{
        // Create semaphores
        semAudio = "semGlobal";
        if (as == asHandsfree) semAudio += "Handsfree";
        else if (as == asHeadset) semAudio += "Headset";
        else if (as == asA2DP) semAudio += "A2DP";

        // Create global mutexes
        semGlobal = CreateSemaphore(NULL,1,1,semAudio.c_str());
        if (semGlobal==NULL)
        {
                lastErr = eeCriticalError;
                lastErrStr = Report("Semaphore creation failed.\n");
                return false;
        }

        // Create udp-sender
        globalvars.udpsender = udpSenderCreate("127.0.0.1",udpOut);
        if (globalvars.udpsender==0)
        {
                lastErr = eeUdpOutInUse;
                lastErrStr = Report((char*)getSocketErrorString(getSendError()).c_str());
                return false;
        }


        // Create UDP receiver
        try
        {
                udprec = new UDPReceiverObject<Caudio>(udpIn,this,&Caudio::udpReceiveCallback);
                int errnr = udprec->getError();
                if (errnr!=0)
                {
                        lastErr = eeUdpInInUse;
                        lastErrStr = Report((char*)getSocketErrorString(errnr).c_str());
                        return false;
                }
                // create and start the thread
                globalvars.udpThreadRec = new Thread(udprec);
                globalvars.udpThreadRec->start();
        }
        catch (ThreadException &e)
        {
                lastErr = eeUdpInInUse;
                lastErrStr = Report((char*)e.Message.c_str());
                return false;
        }

        // Create onWaveInFull and onWaveOutEmpty Threads
        WaveOutEmptyThreadHandle = CreateThread(NULL, 0, &Caudio::WaveOutEmptyThread, &globalvars, 0, &WaveOutEmptyThreadID);
        WaveInFullThreadHandle = CreateThread(NULL, 0, &Caudio::WaveInFullThread, &globalvars, 0, &WaveInFullThreadID);

        if (WaveOutEmptyThreadHandle==NULL || WaveInFullThreadHandle==NULL)
        {
                lastErr = eeCriticalError;
                lastErrStr = Report("WaveIn or/and WaveOut thread creation failed.\n");
                return false;
        }

        CloseHandle(WaveOutEmptyThreadHandle);
        CloseHandle(WaveInFullThreadHandle);

        globalvars.initilized = true;

        return true;
}

void Caudio::getLastError(ErrEnums *err, string *errStr)
{
        *err = lastErr;
        errStr->assign(lastErrStr);
}

Caudio::~Caudio()
{
        stop();
        delete udprec;
        delete globalvars.udpThreadRec;
}

bool Caudio::start()
{
        // http://icculus.org/SDL_sound/downloads/external_documentation/wavecomp.htm

        if (!globalvars.initilized)
        {
                lastErr = eeNotInitilized;
                lastErrStr = Report("Class not initilized. Call initialize first.");
                return false;
        }

        globalvars.delay = true;
        globalvars.numWaveInBuffers = 0;
        globalvars.numWaveOutBuffers = 0;
        globalvars.closing = false;

        MMRESULT mmRC;

        WAVEFORMATEX   wfex;
        wfex.wFormatTag = WAVE_FORMAT_PCM;
        wfex.nChannels = CHANNELS;
        wfex.nSamplesPerSec = SAMPLE_RATE;
        wfex.nAvgBytesPerSec = AVG_BYTES_PER_SEC;
        wfex.nBlockAlign = 2;
        wfex.wBitsPerSample = BITS_PER_SAMPLE;
        wfex.cbSize = 0;
        LPWAVEFORMATEX wfx = &wfex;


        UINT deviceIdIn = WAVE_MAPPER;
        UINT deviceIdOut = WAVE_MAPPER;
        UINT waveMapped = CALLBACK_THREAD;
#ifndef DONT_USE_BLUETOOTH
        waveMapped = CALLBACK_THREAD|WAVE_MAPPED;
        if (!getBluetoothDeviceId(&deviceIdIn,&deviceIdOut))
        {
                lastErr = eeBluetoothDeviceNotFound;
                lastErrStr = Report("Unable to find bluetooth-device!\n");
                return false;
        }
#endif

        // Open wave out device
        mmRC = waveOutOpen(&globalvars.hWaveOut, deviceIdOut, (LPWAVEFORMATEX)wfx, (DWORD)WaveOutEmptyThreadID, 0, waveMapped);

        if (mmRC != MMSYSERR_NOERROR)
        {
                lastErr = eeFailedOpenDevice;
                lastErrStr = Report("Error opening wave playback device: ");
                Report(Merr(mmRC).c_str());
                return false;
        }

        // Open wave in device
        mmRC = waveInOpen(&globalvars.hWaveIn, deviceIdIn, (LPWAVEFORMATEX)wfx, (DWORD)WaveInFullThreadID, 0, waveMapped);

        if (mmRC != MMSYSERR_NOERROR)
        {
                lastErr = eeFailedOpenDevice;
                lastErrStr = Report("Error opening wave input device: ");
                Report(Merr(mmRC).c_str());
                return false;
        }
        else
        {
                waveInStart(globalvars.hWaveIn);
        }

        // Add all inbuffers
        for (int i = 0; i < NUM_BLOCKS; i++)
        {
                // prepare and add blocks to capture device queue
                globalvars.inBlocks[i].Prepare(globalvars.hWaveIn);
                globalvars.inBlocks[i].Add(globalvars.hWaveIn);
                globalvars.numWaveInBuffers++;
        }

        // Setup free list for outbuffers
        for (int i = 0; i < NUM_BLOCKS*2; i++)
        {
                globalvars.outBlocks[i].Prepare(globalvars.hWaveOut);
                globalvars.freeOutList.push(&(globalvars.outBlocks[i]));
        }

        globalvars.started = true;

        return true;
}

bool Caudio::stop()
{
        if (!globalvars.initilized)
        {
                lastErr = eeNotInitilized;
                lastErrStr = "Class not initilized. Call initialize first.";
                return false;
        }

        globalvars.closing = true;

        if (globalvars.hWaveOut != 0)
        { // do if playback device is open
                // Reset playback and close if all buffers are returned
                waveOutReset(globalvars.hWaveOut);
                // Needed because we can be in delay mode
                if (globalvars.numWaveOutBuffers == 0) waveOutClose(globalvars.hWaveOut);
        }

        if (globalvars.hWaveIn != 0)
        { // do if capture device is open
                waveInReset(globalvars.hWaveIn);
                if (globalvars.numWaveInBuffers == 0) waveInClose(globalvars.hWaveIn);
        }

        globalvars.started = false;

        return true;
}

void Caudio::managePlaylist(GlobalVariables *global)
{
        // Add buffers to WaveOut From buffer
        if (global->delay)
        {
                // Wait until playback buffer has at least UPPER_THRESHOLD
                if (global->playbackList.size() >= UPPER_THRESHOLD)
                {
                        // Treshold reached
                        Caudio::Report("Delay off\n");
                        global->delay = false;
                }
        }
        else
        {

                // Add all new WoutBuffer:s from the queue

                unsigned int playbackListSize = (unsigned int)global->playbackList.size();

                for (unsigned int i=0;i<playbackListSize;i++)
                {
                        // Get buffer first in queue
                        Caudio::WoutBuffer *woutbuff = (Caudio::WoutBuffer*)global->playbackList.front();

                        // Prepeare and add to waveout
                        MMRESULT r = woutbuff->Prepare(global->hWaveOut);
                        woutbuff->Add(global->hWaveOut);
                        global->numWaveOutBuffers++;
                        // Remove buffer from playlist
                        global->playbackList.pop();
                }

                if (global->numWaveOutBuffers<=LOWER_THRESHOLD)
                {
                        // Buffer underrun.
                        // delay and catch-up
                        Caudio::Report("Delay on\n");
                        global->delay = true;
                }
        }
```

```
}

bool Caudio::getBluetoothDeviceId(UINT *idIn, UINT *idOut)
{
        WAVEOUTCAPS capsOut;
        WAVEINCAPS capsIn;

        bool found = false;
        for (UINT i=0;i<waveOutGetNumDevs();i++)
        {
                if (waveOutGetDevCaps((UINT_PTR)i,&capsOut,sizeof(WAVEOUTCAPS))==MMSYSERR_NOERROR)
                {
                        string name(capsOut.szPname);

                        if (name.find("luetooth")!=string::npos)
                        {
                                *idOut = i;
                                found = true;
                                break;
                        }
                }
        }

        for (UINT i=0;i<waveInGetNumDevs();i++)
        {
                if (waveInGetDevCaps((UINT_PTR)i,&capsIn,sizeof(WAVEINCAPS))==MMSYSERR_NOERROR)
                {
                        string name(capsIn.szPname);

                        if (name.find("luetooth")!=string::npos)
                        {
                                *idIn = i;
                                return true;
                        }
                }
        }

        return false;
}



string Caudio::Merr(MMRESULT m)
{
        switch (m)
        {
                case MMSYSERR_ALLOCATED: return "Specified resource is already allocated.";
                case MMSYSERR_BADDEVICEID: return "Specified device identifier is out of range.";
                case MMSYSERR_NOERROR: return "success.";
                case MMSYSERR_INVALHANDLE: return "Specified device handle is invalid.";
                case MMSYSERR_NODRIVER: return "No device driver is present.";
                case MMSYSERR_NOMEM: return "Unable to allocate or lock memory.";
                case WAVERR_STILLPLAYING: return "The buffer pointed to by the pwh parameter is still in the queue.";
                case WAVERR_BADFORMAT: return "Attempted to open with an unsupported waveform-audio format.";
                case WAVERR_SYNC: return "The device is synchronous but waveOutOpen was called without using the WAVE_ALLOWSYNC flag.";
                case WAVERR_UNPREPARED: return "The buffer pointed to by the pwh parameter hasn't been prepared.";
        }
        return "Unkown result.";
}

string Caudio::getSocketErrorString(int error)
{
        string errstr;

                switch(error)
                {
                        case WSANOTINITIALISED: errstr = "WSANOTINITIALISED"; break;
                        case WSAENETDOWN: errstr = "WSAENETDOWN"; break;
                        case WSAHOST_NOT_FOUND: errstr = "WSAHOST_NOT_FOUND"; break;
                        case WSATRY_AGAIN: errstr = "WSATRY_AGAIN"; break;
                        case WSANO_RECOVERY: errstr = "WSANO_RECOVERY"; break;
                        case WSANO_DATA: errstr = "WSANO_DATA"; break;
                        case WSAEINPROGRESS: errstr = "WSAEINPROGRESS"; break;
                        case WSAEFAULT: errstr = "WSAEFAULT"; break;
                        case WSAEINTR: errstr = "WSAEINTR"; break;
                        case WSAEMFILE: errstr = "WSAEMFILE"; break;
                        case WSAENOBUFS: errstr = "WSAENOBUFS"; break;
                        case WSAEPROTONOSUPPORT: errstr = "WSAEPROTONOSUPPORT"; break;
                        case WSAEPROTOTYPE: errstr = "WSAEPROTOTYPE"; break;
                        case WSAESOCKTNOSUPPORT: errstr = "WSAESOCKTNOSUPPORT"; break;
                        case WSAEACCES: errstr = "WSAEACCES"; break;
                        case WSAEADDRINUSE: errstr = "WSAEADDRINUSE"; break;
                        case WSAEADDRNOTAVAIL: errstr = "WSAEADDRNOTAVAIL"; break;
                        case WSAEINVAL: errstr = "WSAEINVAL"; break;
                        case WSAENOTSOCK: errstr = "WSAENOTSOCK"; break;
                default: errstr = "default";
                }

                return errstr;
}

void Caudio::udpReceiveCallback(char * data, int size)
{
        // Get global variables mutex
        HANDLE semGlobal = CreateSemaphore(NULL,0,1,semAudio.c_str());
        DWORD semResult = WaitForSingleObject(semGlobal,INFINITE);

        if (!globalvars.started)
        {
                ReleaseSemaphore(semGlobal,1,NULL);
                return;
        }


        // if closing, ignore data
        if (globalvars.closing)
        {
                ReleaseSemaphore(semGlobal,1,NULL);
                return;
        }

        // Free buffers?
        if (!globalvars.freeOutList.empty())
        {
                // DEBUG TODO only add to playlist when not wout is to overull
                if (globalvars.numWaveOutBuffers<MAX_WOUT_BUFFERS)
                {
                        // Get a free buffer
                        WoutBuffer * woutbuffer = globalvars.freeOutList.front();

                        // Copy data to playback buffer
                        memcpy(woutbuffer->m_Data,data,BLOCK_SIZE);
                        globalvars.playbackList.push(woutbuffer);

                        // Remove buffer from free list
                        globalvars.freeOutList.pop();

                        // Manage playlist
                        Caudio::managePlaylist(&globalvars);
                }
        }
        else Report("No free buffers.\n");


        ReleaseSemaphore(semGlobal,1,NULL);
}

DWORD WINAPI Caudio::WaveInFullThread(LPVOID lpParam)
{
        MSG             msg;

        Caudio::GlobalVariables *globalvars = (Caudio::GlobalVariables*)lpParam;

        while (GetMessage(&msg, 0, 0, 0) == 1)
        {
                /* Figure out which message was sent */
                switch (msg.message)
                {
                        case MM_WIM_DATA:
                                // Get global variables mutex
                                HANDLE semGlobal = CreateSemaphore(NULL,0,1,semAudio.c_str());
                                DWORD semResult = WaitForSingleObject(semGlobal,INFINITE);

                                WAVEHDR* pWHDR = (WAVEHDR*)msg.lParam;
                                Caudio::WinBuffer * wb = (Caudio::WinBuffer*)(pWHDR->dwUser);

                                // Unprepare it and add it to udp out list
                                wb->Unprepare(globalvars->hWaveIn);
                                globalvars->numWaveInBuffers--;

                                // Send buffer
                                if (wb!=NULL && !globalvars->closing)
                                {
                                        udpSenderSend(globalvars->udpsender,(const char *)wb->m_Data,BLOCK_SIZE);
                                        wb->Prepare(globalvars->hWaveIn);
                                        wb->Add(globalvars->hWaveIn);
                                        globalvars->numWaveInBuffers++;
                                }
                                /*
                                printf("udp in: %d, udp out: %d, free: %d, play: %d, win: %d, wout: %d\n",
                                globalvars->udpPckReceived,
                                        globalvars->udpPckSent,
                                        globalvars->freeOutList.size(),
                                        globalvars->playbackList.size(),
                                        globalvars->numWaveInBuffers,
                                        globalvars->numWaveOutBuffers);
                                */

                                if (globalvars->closing) waveInClose(globalvars->hWaveIn);

                                ReleaseSemaphore(semGlobal,1,NULL);
                        break;
                }
        }

        ExitThread(0);
    return 1;
}
```

```
DWORD WINAPI Caudio::WaveOutEmptyThread(LPVOID lpParam)
{
        MSG             msg;

        Caudio::GlobalVariables *globalvars = (Caudio::GlobalVariables*)lpParam;

        while (GetMessage(&msg, 0, 0, 0) == 1)
        {
                /* Figure out which message was sent */
                switch (msg.message)
                {
                        case MM_WOM_DONE:
                                // Get global variables mutex
                                HANDLE semGlobal = CreateSemaphore(NULL,0,1,semAudio.c_str());
                                DWORD semResult = WaitForSingleObject(semGlobal,INFINITE);

                                WAVEHDR* pWHDR = (WAVEHDR*)msg.lParam;
                                Caudio::WoutBuffer * wb = (Caudio::WoutBuffer*)(pWHDR->dwUser);

                                // Unprepare it and add it to the free list
                                wb->Unprepare(globalvars->hWaveOut);
                                globalvars->numWaveOutBuffers--;
                                globalvars->freeOutList.push(wb);

                                // manage playlist
                                if (!globalvars->closing) Caudio::managePlaylist(globalvars);
                                else if (globalvars->numWaveOutBuffers<0) waveOutClose(globalvars->hWaveOut);
                                ReleaseSemaphore(semGlobal,1,NULL);
                        break;
                }
        }

        ExitThread(0);
    return 1;

}
```

14.4.3 File main.cpp

```
#include "stdafx.h"
#include "audio.h"

int _tmain(int argc, _TCHAR* argv[])
{
Caudio *cad = new Caudio();
cad->initialize(9102,9103,Caudio::asHandsfree); // The last argument does not matter when running
// as stand alone.
cad->start();

getchar();
cad->stop();

return 0;
}
```