TeamSpeak 3 Client
SDK Developer Manual

Revision 9935

Table of Contents

Copyright 2
License agreement 2
Introduction 5
Architecture overview 5
System requirements 7
Overview of header files 7
Calling Client Lib functions 8
Return code 9
Initializing 9
The callback mechanism 10
Querying the library version 11
Shutting down 12
Managing server connection handlers 12
Connecting to a server 13
Disconnecting from a server 18
Error handling 19
Logging 21
User-defined logging 22
Using playback and capture modes and devices 23
Initializing modes and devices 24
Querying available modes and devices 25
Checking current modes and devices 29
Closing devices 30
Activating the capture device 31
Custom FMOD objects 32
Using custom FMOD system objects 32
Customizing FMOD channel objects 35
Querying the current FMOD System objects 36
Sound codecs 37
Encoder options 38
Preprocessor options 39
Playback options 42
3D Sound 45
Query available servers, channels and clients 47
Retrieve and store information 51
Client information 51
Information related to own client 51
Information related to other clients 56
Whisper lists 58
Channel information 59

Channel sorting 65
Server information 66
Interacting with the server 69
Joining a channel 69
Creating a new channel 72
Deleting a channel 74
Moving a channel 75
Text chat 77
Sending 77
Receiving 78
Kicking clients 79
Channel subscriptions 81
Muting clients locally 84
Custom encryption 85
Other events 86
Voice recording 89
Miscellaneous functions 90
FAQ 91
Revision history 93
Index 94

Copyright

TeamSpeak Systems GmbH
Soiernstrasse 1
82494 Krün
Germany

Visit TeamSpeak-Systems on the web at www.teamspeak-systems.de [http://www.teamspeak.com-systems.de]

License agreement

TeamSpeak 3

LICENSE AGREEMENT

October 25th, 2007

THIS IS A LEGAL AGREEMENT between "you," the company or end user of TeamSpeak 3 brand software, and TeamSpeak Systems GmbH, a Krün, Germany company hereafter referred to as "TeamSpeak Systems".

Use of the software you are about to install indicates your acceptance of these terms. You also agree to accept these terms by so indicating at the appropriate screen, prior to the download or installation process. As used in this Agreement, the capitalized term "Software" means the TeamSpeak 3 voice over IP (VoIP) communication software together with any and all enhancements, upgrades, and updates that may be provided to you in the future by TeamSpeak Systems. IF YOU DO NOT AGREE TO THESE TERMS AND CONDITIONS, YOU SHOULD SO INDICATE BY CONTACTING TEAMSPEAK SYSTEMS AND PROMPTLY DISCONTINUE THE INSTALLATION PROCESS AND USE OF THIS SOFTWARE.

Ownership

The Software and any accompanying documentation are owned by TeamSpeak Systems and ownership of the Software shall at all times remain with TeamSpeak Systems. Copies are provided to you only to allow you to exercise your rights under this Agreement. This Agreement does not constitute a sale of the Software or any accompanying documentation, or any portion thereof. Without limiting the generality of the foregoing, you do not receive any rights to any patents, copyrights, trade secrets, trademarks or other intellectual property rights relating to or in the Software or any accompanying documentation. All rights not expressly granted to you under this Agreement are reserved by TeamSpeak Systems.

Grant of License Applicable To TeamSpeak 3

Subject to the terms and conditions set out in this Agreement, TeamSpeak Systems grants you a limited, nonexclusive, non-transferable and nonsublicensable right to use the Software called "TeamSpeak 3" solely in accordance with the following terms and conditions:

1. Use of TeamSpeak 3. You may use TeamSpeak 3 on multiple computers owned, leased or rented by you, your company, or business entity; however, you are the only individual, company, or business entity with the right to use your licensed copy(ies) of TeamSpeak 3. All copies of TeamSpeak 3 must include TeamSpeak Systems' copyright notice.

2. Distribution Prohibited. You may not distribute copies of TeamSpeak 3 for use by anyone other than you, your company, or business entity. Distribution of TeamSpeak 3 by you to third parties is hereby expressly prohibited.

3. Fees. As of the date listed above for this License Agreement, TeamSpeak 3 is in a "pre-release" stage. Fees and licensing costs will be determined when the final version of the product is released or an agreed upon commencement date for commercial use of the Software is initiated.

4. Termination. TeamSpeak Systems may terminate your TeamSpeak 3 license at any time, for any reason or no reason. TeamSpeak Systems may also terminate your TeamSpeak 3 license if you breach any of the terms and conditions set forth in this Agreement. Upon termination, you shall immediately destroy all copies of TeamSpeak 3 and any accompanying files or documentation in your possession, custody or control.

5. Support. TeamSpeak Systems will provide you with support services related to TeamSpeak 3 for a period that begins on the date TeamSpeak 3 is delivered to you, and ends upon the termination of this Agreement.

6. Upgrades. TeamSpeak Systems will provide you with upgrades to TeamSpeak 3 for a period that begins on the date TeamSpeak 3 is delivered to you. Such upgrades will be released only by TeamSpeak Systems for the purpose of improving TeamSpeak 3 software. TeamSpeak Systems has no obligation to provide you with any upgrades that are not released for general distribution to TeamSpeak Systems' other licensees. Nothing in this Agreement shall be construed to obligate TeamSpeak Systems to provide upgrades to you under any circumstances.

Prohibited Conduct

You represent and warrant that you will not violate any of the terms and conditions set forth in this Agreement and that:

You will not, and will not permit others to: (i) reverse engineer, decompile, disassemble, derive the source code of, modify, or create derivative works from the Software; or (ii) use, copy, modify, alter, or transfer, electronically or otherwise, the Software or any of the accompanying documentation except as expressly permitted in this Agreement; or (iii) redistribute, sell, rent, lease, sublicense, or otherwise transfer rights to the Software whether in a stand-alone configuration or as incorporated with other software code written by any party except as expressly permitted in this Agreement.

You will not use the Software to engage in or allow others to engage in any illegal activity.

You will not engage in use of the Software that will interfere with or damage the operation of the services of third parties by overburdening/disabling network resources through automated queries, excessive usage or similar conduct.

You will not use the Software to engage in any activity that will violate the rights of third parties, including, without limitation, through the use, public display, public performance, reproduction, distribution, or modification of communications or materials

that infringe copyrights, trademarks, publicity rights, privacy rights, other proprietary rights, or rights against defamation of third parties.

You will not transfer the Software or utilize the Software in combination with third party software authored by you or others to create an integrated software program which you transfer to unrelated third parties.

Upgrades, Updates And Enhancements

All upgrades, updates or enhancements of the Software shall be deemed to be part of the Software and will be subject to this Agreement.

Disclaimer of Warranty

THE SOFTWARE IS PROVIDED ON AN "AS IS" BASIS, WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE WARRANTIES THAT IT IS FREE OF DEFECTS, VIRUS FREE, ABLE TO OPERATE ON AN UNINTERRUPTED BASIS, MERCHANTABLE, FIT FOR A PARTICULAR PURPOSE OR NON-INFRINGING. THIS DISCLAIMER OF WARRANTY CONSTITUTES AN ESSENTIAL PART OF THIS LICENSE AND AGREEMENT. NO USE OF THE SOFTWARE IS AUTHORIZED HEREUNDER EXCEPT UNDER THIS DISCLAIMER.

Limitation of Liability

TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT WILL TEAMSPEAK SYSTEMS BE LIABLE FOR ANY INDIRECT, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOST PROFITS, LOSS OF GOODWILL, WORK STOPPAGE, COMPUTER FAILURE OR MALFUNCTION, OR ANY AND ALL OTHER COMMERCIAL DAMAGES OR LOSSES, EVEN IF ADVISED OF THE POSSIBILITY THEREOF, AND REGARDLESS OF THE LEGAL OR EQUITABLE THEORY (CONTRACT, TORT OR OTHERWISE) UPON WHICH THE CLAIM IS BASED. IN ANY CASE, TEAMSPEAK SYSTEMS' COLLECTIVE LIABILITY UNDER ANY PROVISION OF THIS LICENSE SHALL NOT EXCEED IN THE AGGREGATE THE SUM OF THE FEES (IF ANY) YOU PAID FOR THIS LICENSE.

Legends and Notices

You agree that you will not remove or alter any trademark, logo, copyright or other proprietary notices, legends, symbols or labels in the Software or any accompanying files or documentation.

Term and Termination

This Agreement is effective upon your acceptance as provided herein and payment of the applicable license fees (if any), and will remain in force until terminated. You may terminate the licenses granted in this Agreement at any time by contacting TeamSpeak Systems in writing, and destroying the Software and any accompanying files or documentation, together with any and all copies thereof. The licenses granted in this Agreement will terminate automatically if you breach any of its terms or conditions or any of the terms or conditions of any other agreement between you and TeamSpeak Systems. Upon termination, you shall immediately destroy the original and all copies of the Software and any accompanying documentation, or return them to TeamSpeak Systems.

Software Suggestions

TeamSpeak Systems welcomes suggestions for enhancing the Software and any accompanying documentation that may result in computer programs, reports, presentations, documents, ideas or inventions relating or useful to TeamSpeak Systems' business. You acknowledge that all title, ownership rights, and intellectual property rights concerning such suggestions shall become the exclusive property of TeamSpeak Systems and may be used for its business purposes in its sole discretion without any payment or accounting to you.

Miscellaneous

This Agreement constitutes the entire agreement between the parties concerning the Software, and may be amended only by a writing signed by both parties. This Agreement shall be governed by the laws of Krün, Germany, excluding its conflict of law provisions. All disputes relating to this Agreement are subject to the exclusive jurisdiction of the courts within Germany and you expressly consent to the exercise of personal jurisdiction in the courts of Germany in connection with any such dispute. This Agreement shall not be governed by the United Nations Convention on Contracts for the International Sale of Goods. If any provision in this Agreement should be held illegal or unenforceable by a court of competent jurisdiction, such provision shall be modified to the extent necessary to render it enforceable without losing its intent, or severed from this Agreement if no such modification is possible, and other provisions of this Agreement shall remain in full force and effect. A waiver by either party of any term or condition of this Agreement or any breach thereof, in any one instance, shall not waive such term or condition or any subsequent breach thereof.

Introduction

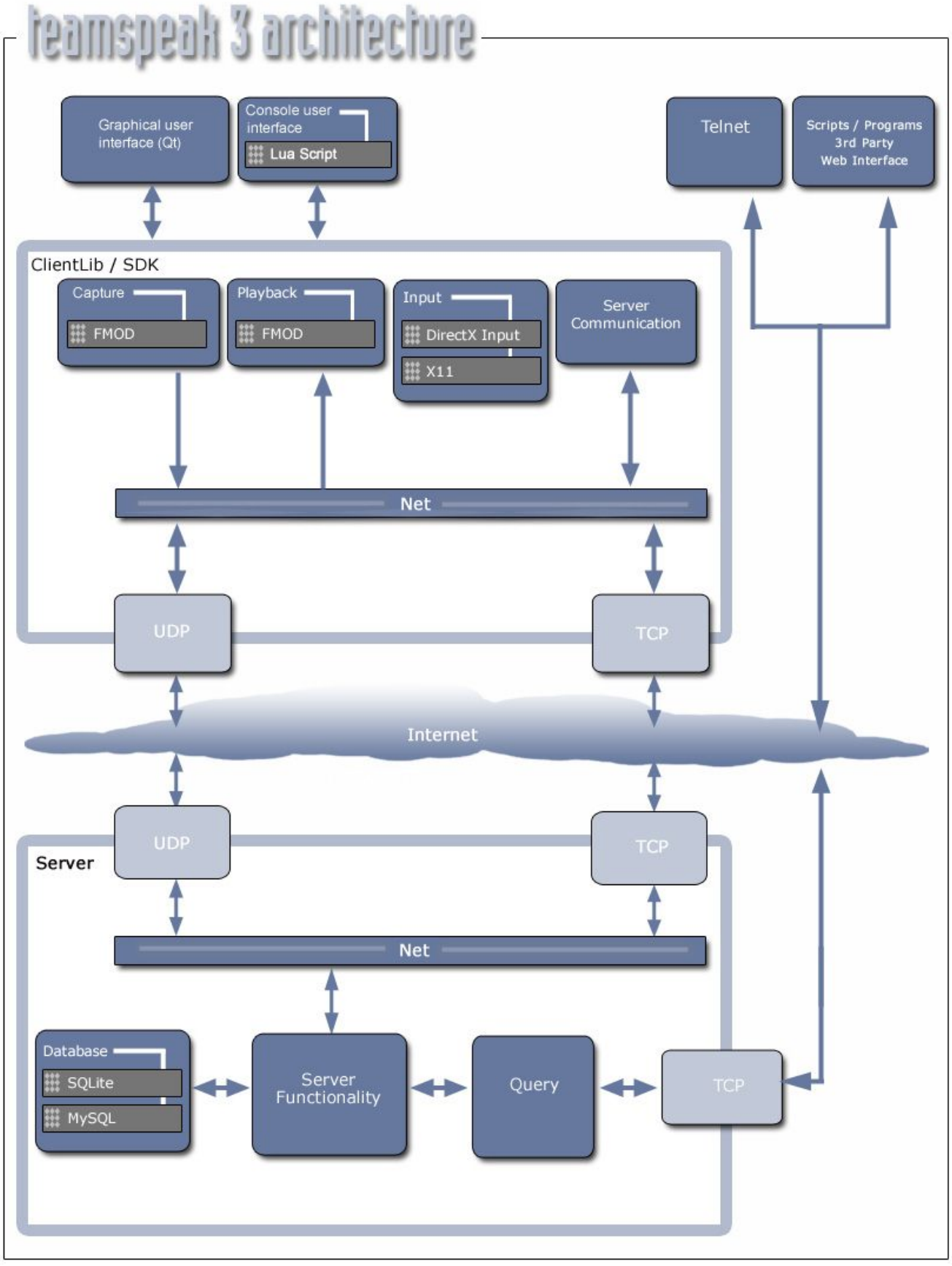
TeamSpeak 3 is the successor of the popular TeamSpeak 2 software, a scalable Voice-Over-IP application consisting of client and server software. TeamSpeak is generally regarded as the leading VoIP system offering a superior voice quality, scalability and usability.

TeamSpeak 3 aims at meeting the high expectations while offering more possibilities to both endusers and third-party developers.

Beginning with an overview of the TeamSpeak 3 architecture, this document provides an introduction to client-side programming with the TeamSpeak 3 SDK, the so-called Client Lib. This library encapsulates client-side functionality while keeping the user interface separated and modular.

Architecture overview

The following chart presents a high-level overview of the TeamSpeak 3 architecture.


teamspeak 3 architecture
Graphical user interface (Qt)
Console user interface
Lua Script
Telnet
Scripts / Programs 3rd Party Web Interface
ClientLib / SDK
Capture
FMOD
Playback
FMOD
Input
DirectX Input
X11
Server Communication
Net
UDP
TCP
Internet
UDP
TCP
Server
Net
Database
SQLite
MySQL
Server Functionality
Query
TCP

- User Interface

  The part of the application you are developing. The client is the user front-end, which offers the interface to connect to a TeamSpeak 3 server and communicate with other users.

  The TeamSpeak 3 client developed by TeamSpeak Systems utilizes Qt as a cross-platform GUI library to allow a rich client interface for Windows, Linux and Mac OS X. In addition, a small console client scriptable with Lua assists in development and testing tasks.

- Client Lib / SDK

  The Client Lib is responsible for all communication between Client UI and Server. The Client UI will exclusively call Client Lib functions and never interact with the Server directly. Vice versa, the Client Lib handles traffic sent by the server and forwards events to the Client UI for displaying to the enduser.

  Reponsible for sound input and ouput, the Client Lib uses industry-standard libraries like FMOD, Speex (all platforms), DirectSound (Windows), CoreAudio (Macintosh) and Alsa (Linux). Speech preprocessing, encoding and decoding takes place in this package.

  Splitting the client-side functionality and interface allows to exchange the Client UI with a customized replacement. This flexible modular architecture offers the possibility to add third-party interfaces to the TeamSpeak 3 network.

- Server

  The TeamSpeak 3 server is mainly responsible for routing the network traffic from and to the connected clients. The server SDK allows to develop extensions to add customized functionality.

System requirements

For developing third-party clients with the TeamSpeak 3 Client Lib the following system requirements apply:

- Windows

  Windows 2000, XP, Vista (32- and 64-bit)

- Mac OS X

  Mac OS X 10.3.9, 10.4, 10.5 on Intel and PowerPC Macs

- Linux

  Any recent Linux distribution with libstdc++ 6. Both 32- and 64-bit are supported.

  Developed and tested on Gentoo and Ubuntu 7.04, 7.10 and 8.04.

Important

The calling convention used in the functions exported by the shared TeamSpeak 3 SDK libaries is *cdecl*. You must not use another calling convention, like stdcall on Windows, when declaring function pointers to the TeamSpeak 3 SDK libraries. Otherwise stack corruption at runtime may occur.

Overview of header files

The following header files are deployed to SDK developers:

- `clientlib.h`

  Declares the function prototypes and callbacks for the communication between Client Lib and Client UI. While the Client UI makes function calls into the Client Lib using the declared prototypes, the Client Lib calls the Client UI via callbacks.

- `clientlib_publicdefinitions.h`

  Defines various enums and structs used by the Client UI and Client Lib. These definitions are used by the functions and callbacks declared in `clientlib.h`

- `public_definitions.h`

  Defines various enums and structs used by both client- and server-side.

- `public_errors.h`

  Defines the error codes returned by every Client Lib function and `onServerErrorEvent`. Error codes are organized in several groups. The first byte of the error code defines the error group, the second the count within the group.

Calling Client Lib functions

Client Lib functions follow a common pattern. They always return an error code or *`ERROR_ok`* on success. If there is a result variable, it is always the last variable in the functions parameters list.

```
ERROR ts3client_FUNCNAME(arg1, arg2, ..., &result);
```

Result variables should *only* be accessed if the function returned *`ERROR_ok`*. Otherwise the state of the result variable is undefined.

In those cases where the result variable is a basic type (int, float etc.), the memory for the result variable has to be declared by the caller. Simply pass the address of the variable to the Client Lib function.

```
int result;

if(ts3client_XXX(arg1, arg2, ..., &result) == ERROR_ok) {
    /* Use result variable */
} else {
    /* Handle error, result variable is undefined */
}
```

If the result variable is a pointer type (C strings, arrays etc.), the memory is allocated by the Client Lib function. In that case, the caller has to release the allocated memory later by using `ts3client_freeMemory`. It is important to *only* access and release the memory if the function returned *`ERROR_ok`*. Should the function return an error, the result variable is uninitialized, so freeing or accessing it could crash the application.

```
char* result;

if(ts3client_XXX(arg1, arg2, ..., &result) == ERROR_ok) {
    /* Use result variable */
    ts3client_freeMemory(result);  /* Release result variable */
} else {
    /* Handle error, result variable is undefined. Do not access or release it. */
}
```

Note

Client Lib functions are *thread-safe*. It is possible to access the Client Lib from several threads at the same time.

Return code

Client Lib functions that interact with the server take an additional parameter *returnCode*, which can be used to find out which action results in a later server error. If you pass a custom string as return code, the onServerErrorEvent callback will receive the same custom string in its *returnCode* parameter. If no error occured, onServerErrorEvent will indicate success py passing the error code *ERROR_ok*.

Pass NULL as *returnCode* if you do not need the feature. In this case, if no error occurs onServerErrorEvent will *not* be called.

An example, request moving a client:

```
ts3client_requestClientMove(scHandlerID, clientID, newChannelID, password, "MyClientMoveReturnCode");
```

If an error occurs, the onServerErrorEvent callback is called:

```
void my_onServerErrorEvent(anyID serverConnectionHandlerID, const char* errorMessage,
                           unsigned int error, const char* returnCode, const char* extraMessage) {
    if(strcmp(returnCode, "MyClientMoveReturnCode")) == 0) {
        /* We know this error is the reaction to above called function as we got the same returnCode */
 if(error == ERROR_ok) {
     /* Success */
 }
}
```

Initializing

When starting the client, initialize the Client Lib with a call to

```
unsigned int ts3client_initClientLib(functionPointers, functionRarePointers, usedLogTypes, logFileFolder);
```

```
const struct ClientUIFunctions* functionPointers;
const struct ClientUIFunctionsRare* functionRarePointers;
int usedLogTypes;
const char* logFileFolder;
```

Parameters

- *functionPointers*

  Callback function pointers. See below.

- *functionRarePointers*

  Unused by SDK, pass NULL.

- *usedLogTypes*

  Defines the log output types. The Client Lib can output log messages (called by ts3client_logMessage) to a file (located in the logs directory relative to the client executable), to stdout or to user defined callbacks. If user callbacks are activated, the onUserLoggingMessageEvent event needs to be implemented.

  Available values are defined by the enum LogTypes (see public_definitions.h):

```
enum LogTypes {
    LogType_NONE          = 0x0000,
    LogType_FILE          = 0x0001,
    LogType_CONSOLE       = 0x0002,
    LogType_USERLOGGING   = 0x0004,
    LogType_NO_NETLOGGING = 0x0008,
    LogType_DATABASE      = 0x0010,
};
```

Multiple log types can be combined with a binary OR. If only *LogType_NONE* is used, local logging is disabled.

Note

Logging to console can slow down the application on Windows. Hence we do not recommend to log to the console on Windows other than in debug builds.

Note

If *LogType_NO_NETLOGGING* is not passed, the Client Lib will send notifications of each warning, error and critical error to a TeamSpeak-Systems webserver. This data is used for analysis and debugging during the TeamSpeak 3 development.

We recommend to leave netlogging enabled in debug builds.

LogType_DATABASE has no effect in the Client Lib, this is only used by the server.

- *logFileFolder*

  Location where the logfiles produced if file logging is enabled will be saved to. Pass NULL for the default behaviour, which is to use a folder called `logs` in the current working directory.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Note

This function must not be called more than once.

The callback mechanism

The communication from Client Lib to Client UI takes place using callbacks. The Client UI has to define a series of function pointers using the struct ClientUIFunctions (see `clientlib.h`). These callbacks are used to forward any incoming server actions to the Client UI for further processing.

A callback example in C:

```
static void my_onConnectStatusChangeEvent_Callback(anyID serverConnectionHandlerID,
                                                   int newStatus,
                                                   int errorNumber) {
    /* Implementation */
}
```

C++ developers can also use static member functions for the callbacks.

Before calling `ts3client_initClientLib`, create an instance of struct ClientUIFunctions, initialize all function pointers with NULL and assign the structs function pointers to your callback functions:

```
unsigned int error;

/* Create struct */
ClientUIFunctions clUIFuncs;

/* Initialize all function pointers with NULL */
memset(&clUIFuncs, 0, sizeof(struct ClientUIFunctions));

/* Assign those function pointers you implemented */
clUIFuncs.onConnectStatusChangeEvent = my_onConnectStatusChangeEvent_Callback;
clUIFuncs.onNewChannelEvent          = my_onNewChannelEvent_Callback;
(...)

/* Initialize client lib with callback function pointers */
error = ts3client_initClientLib(&clUIFuncs, NULL, LogType_FILE | LogType_CONSOLE);
if(error != ERROR_ok) {
    printf("Error initializing clientlib: %d\n", error);
    (...)
}
```

Important

As long as you initialize unimplemented callbacks with NULL, the Client Lib won't attempt to call those function pointers. However, if you leave unimplemented callbacks undefined, the Client Lib will crash when trying to calling them.

Note

All callbacks used in the SDK are found in the struct ClientUIFunctions (see `public_definitions.h`). Callbacks bundled in the struct ClientUIFunctionsRare are not used by the SDK. These callbacks were split in a separate struct to avoid polluting the SDK headers with code used only internally.

Querying the library version

The Client Lib version can be queried with

```
unsigned int ts3client_getClientLibVersion(result);

char** result;
```

Parameters

- *result*

  Address of a variable that receives the clientlib version string, encoded in UTF-8.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. If an error occured, the result string is uninitialized and must not be accessed.

Caution

The result string must be released using `ts3client_freeMemory`. If an error has occured, the result string is uninitialized and must not be released.

An example using `ts3client_getClientLibVersion`:

```
unsigned int error;
char* version;
error = ts3client_getClientLibVersion(&version);
if(error != ERROR_ok) {
    printf("Error querying clientlib version: %d\n", error);
    return;
}
printf("Client library version: %s\n", version);  /* Print version */
ts3client_freeMemory(version);  /* Release string */
```

Shutting down

Before exiting the client application, the Client Lib should be shut down with

```
unsigned int ts3client_destroyClientLib();
```

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Make sure to call this function *after* disconnecting from any TeamSpeak 3 servers. Any call to Client Lib functions after shutting down has undefined results.

Managing server connection handlers

Before connecting to a TeamSpeak 3 server, a new server connection handler needs to be spawned. Each handler is identified by a unique ID (usually called *serverConnectionHandlerID*). With one server connection handler a connection can be established and dropped multiple times, so for simply reconnecting to the same or another server no new handler needs to be spawned but existing ones can be reused. However, for using multiple connections simultaneously a new handler has to be spawned for each connection.

To create a new server connection handler and receive its ID, call

```
unsigned int ts3client_spawnNewServerConnectionHandler(port, result);

int port;
anyID* result;
```

Parameters

- *port*

  Port the client should bind on. Specify zero to let the operating system chose any free port. In most cases passing zero is sufficient.

  If *port* is specified, the function return value should be checked for *ERROR_unable_to_bind_network_port*. Handle this error by switching to an alternative port until a "free" port is hit and the function returns *ERROR_ok*.

Caution

Do not specify a non-zero value for *port* unless you absolutely know what you are doing.

- *result*

  Address of a variable that receives the server connection handler ID.

To destroy a server connection handler, call

```
unsigned int ts3client_destroyServerConnectionHandler(serverConnectionHandlerID);
```

```
anyID serverConnectionHandlerID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler to destroy.

Both functions return *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Important

Destroying invalidates the handler ID, so it must not be used anymore afterwards. Also do not destroy a server connection handler ID from within a callback.

Connecting to a server

To connect to a server, a client application is required to request an identity from the Client Lib. This string should be requested only once and then locally stored in the applications configuration. The next time the application connects to a server, the identity should be read from the configuration and reused again.

```
unsigned int ts3client_createIdentity(result);
```

```
char** result;
```

Parameters

- *result*

  Address of a variable that receives the identity string, encoded in UTF-8.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error occured, the result string is uninitialized and must not be accessed.

Caution

The result string must be released using `ts3client_freeMemory`. If an error has occured, the result string is uninitialized and must not be released.

Once a server connection handler has been spawned and an identity is available, connect to a TeamSpeak 3 server with

```
unsigned int ts3client_startConnection(serverConnectionHandlerID, identity, ip, port, nickname, defaultChannelArray, defaultChannelPassword, serverPassword);

anyID serverConnectionHandlerID;
const char* identity;
const char* ip;
unsigned int port;
const char* nickname;
const char** defaultChannelArray;
const char* defaultChannelPassword;
const char* serverPassword;
```

Parameters

- *serverConnectionHandlerID*

  Unique identifier for this server connection. Created with `ts3client_spawnNewServerConnectionHandler`

- *identity*

  The clients identity. This string has to be created by calling `ts3client_createIdentity`. Please note an application should create the identity only once, store the string locally and reuse it for future connections.

- *ip*

  Hostname or IP of the TeamSpeak 3 server.

  If you pass a hostname instead of an IP, the Client Lib will try to resolve it to an IP, but the function may block for an unusually long period of time while resolving is taking place. If you are relying on the function to return quickly, we recommend to resolve the hostname yourself (e.g. asynchronously) and then call `ts3client_startConnection` with the IP instead of the hostname.

- *port*

  UDP port of the TeamSpeak 3 server, by default 9987. TeamSpeak 3 uses UDP. Support for TCP might be added in the future.

- *nickname*

  On login, the client attempts to take this nickname on the connected server. Note this is not necessarily the actually assigned nickname, as the server can modifiy the nickname ("gandalf_1" instead the requested "gandalf") or refuse blocked names.

- *defaultChannelArray*

String array defining the path to a channel on the TeamSpeak 3 server. If the channel exists and the user has sufficient rights and supplies the correct password if required, the channel will be joined on login.

To define the path to a subchannel of arbitrary level, create an array of channel names detailing the position of the default channel (e.g. "grandparent", "parent", "mydefault", ""). The array is terminated with a empty string.

Pass NULL to join the servers default channel.

- *defaultChannelPassword*

  Password for the default channel. Pass an empty string if no password is required or no default channel is specified.

- *serverPassword*

  Password for the server. Pass an empty string if the server does not require a password.

All strings need to be encoded in UTF-8 format.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. When trying to connect with an invalid identity, the Client Lib will set the error *ERROR_client_could_not_validate_identity*.

Example code to request a connection to a TeamSpeak 3 server:

```
unsigned int error;
anyID scHandlerID;
char* identity;

error = ts3client_spawnNewServerConnectionHandler(&scHandlerID);
if(error != ERROR_ok) {
    printf("Error spawning server conection handler: %d\n", error);
    return;
}

error = ts3client_createIdentity(&identity);  /* Application should store and reuse the identity */
if(error != ERROR_ok) {
    printf("Error creating identity: %d\n", error);
    return;
}

error = ts3client_startConnection(scHandlerID,
                                  identity
                                  "my-teamspeak-server.com",
      9987,
      "Gandalf",
      NULL,      // Join servers default channel
      "",        // Empty default channel password
      "secret"); // Server password
if(error != ERROR_ok) {
    (...)
}
ts3client_freeMemory(identity);  /* Don't need this anymore */
```

After calling ts3client_startConnection, the client will be informed of the connection status changes by the callback

```
void onConnectStatusChangeEvent(serverConnectionHandlerID, newStatus, errorNumber);

anyID serverConnectionHandlerID;
```

```
int newStatus;
int errorNumber;
```

Parameters

- *newStatus*

  The new connect state as defined by the enum ConnectStatus:

```
enum ConnectStatus {
    STATUS_DISCONNECTED = 0,       //There is no activity to the server, this is the default value
    STATUS_CONNECTING,             //We are trying to connect, we haven't got a clientID yet, we
                                   //haven't been accepted by the server
    STATUS_CONNECTED,              //The server has accepted us, we can talk and hear and we got a
                                   //clientID, but we don't have the channels and clients yet, we
                                   //can get server infos (welcome msg etc.)
    STATUS_CONNECTION_ESTABLISHING,//we are CONNECTED and we are visible
    STATUS_CONNECTION_ESTABLISHED, //we are CONNECTED and we have the client and channels available
};
```

- *errorNumber*

  Should be *ERROR_ok* (zero) when connecting

While connecting, the states will switch through the values *STATUS_CONNECTING*, *STATUS_CONNECTED* and *STATUS_CONNECTION_ESTABLISHED*. Once the state *STATUS_CONNECTED* has been reached, there the server welcome message is available, which can be queried by the client:

- Welcome message

  Query the server variable *VIRTUALSERVER_WELCOMEMESSAGE* for the message text using the function ts3client_getServerVariableAsString:

```
char* welcomeMsg;
if(ts3client_getServerVariableAsString(serverConnectionHandlerID, VIRTUALSERVER_WELCOMEMESSAGE, &welcomeMsg)
   != ERROR_ok) {
    printf("Error getting server welcome message: %d\n", error);
    return;
}
print("Welcome message: %s\n", welcomeMsg);  /* Display message */
ts3client_freeMemory(welcomeMsg);  /* Release memory */
```

To check if a connection to a given server connection handler is established, call:

```
unsigned int ts3client_getConnectionStatus(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
int* result;
```

Parameters

- *serverConnectionHandlerID*

ID of the server connection handler of which the connection state is checked.

- *result*

  Address of a variable that receives the result: 1 - Connected, 0 - Not connected.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After the state *STATUS_CONNECTED* has been reached, the client is assigned an ID which identifies the client on this server. This ID can be queried with

```
unsigned int ts3client_getClientID(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
anyID* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which we are querying the own client ID.

- *result*

  Address of a variable that receives the client ID. Client IDs start with the value 1.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After connection has been established, all current channels on the server are announced to the client. This happens with delays to avoid a flood of information after connecting. The client is informed about the existance of each channel with the following event:

```
void onNewChannelEvent(serverConnectionHandlerID, channelID, channelParentID);

anyID serverConnectionHandlerID;
anyID channelID;
anyID channelParentID;
```

Parameters

- *serverConnectionHandlerID*

  The server connection handler ID.

- *channelID*

The ID of the announced channel.

- *channelParentID*

  ID of the parent channel.

Channel IDs start with the value 1.

The order in which channels are announced by `onNewChannelEvent` is defined by the channel order as explained in the chapter Channel sorting.

All clients currently logged to the server are announced after connecting with the callback `onClientMoveEvent`.

Disconnecting from a server

To disconnect from a TeamSpeak 3 server call

```
unsigned int ts3client_stopConnection(serverConnectionHandlerID, quitMessage);

anyID serverConnectionHandlerID;
const char* quitMessage;
```

Parameters

- *serverConnectionHandlerID*

  The unique ID for this server connection handler.

- *quitMessage*

  A message like for example "leaving". The string needs to be encoded in UTF-8 format.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Like with connecting, on successful disconnecting the client will receive an event:

```
void onConnectStatusChangeEvent(serverserverConnectionHandlerID, newStatus, error-
Number);

anyID serverserverConnectionHandlerID;
int newStatus;
int errorNumber;
```

Parameters

- *newStatus*

Set to *STATUS_DISCONNECTED* as defined by the enum ConnectStatus.

- *errorNumber*

  errorNumber is expected to be *ERROR_ok* as response to calling ts3client_stopConnection.

  Values other than *ERROR_ok* occur when the connection has been lost for reasons not initiated by the user, e.g. network error, forcefully disconnected etc.

Should the server be shutdown, the follow event will be called:

```
void onServerStopEvent(serverConnectionHandlerID, shutdownMessage);

anyID serverConnectionHandlerID;
const char* shutdownMessage;
```

Parameters

- *serverConnectionHandlerID*

  Server connection handler ID of the stopped server.

- *shutdownMessage*

  Message announcing the reason for the shutdown sent by the server. Has to be encoded in UTF-8 format.

Error handling

Each Client Lib function returns either *ERROR_ok* on success or an error value as defined in public_errors.h if the function fails.

The returned error codes are organized in groups, where the first byte defines the error group and the second the count within the group: The naming convention is ERROR_<group>_<error>, for example *ERROR_client_invalid_id*.

Example:

```
unsigned int error;
char* welcomeMsg;

error = ts3client_getServerVariableAsString(serverConnectionHandlerID,
                                            VIRTUALSERVER_WELCOMEMESSAGE,
                                            &welcomeMsg);
if(error != ERROR_ok) {
    /* Handle error */
    return;
}
/* Use welcomeMsg... */
ts3client_freeMemory(welcomeMsg);  /* Release memory *only* if function did not return an error */
```

Note

Result variables should *only* be accessed if the function returned *ERROR_ok*. Otherwise the state of the result variable is undefined.

Important

Some Client Lib functions dynamically allocate memory which has to be freed by the caller using `ts3client_freeMemory`. It is important to *only* access and release the memory if the function returned *`ERROR_ok`*. Should the function return an error, the result variable is uninitialized, so freeing or accessing it could crash the application.

See the section Calling Client Lib functions for additional notes and examples.

A printable error string for a specific error code can be queried with

```
unsigned int ts3client_getErrorMessage(errorCode, error);

unsigned int errorCode;
char** error;
```

Parameters

- *`errorCode`*

  The error code returned from all Client Lib functions.

- *`error`*

  Address of a variable that receives the error message string, encoded in UTF-8 format. Unless the return value of the function is not *`ERROR_ok`*, the string should be released with `ts3client_freeMemory`.

Example:

```
unsigned int error;
anyID myID;

error = ts3client_getClientID(scHandlerID, &myID);  /* Calling some Client Lib function */
if(error != ERROR_ok) {
    char* errorMsg;
    if(ts3client_getErrorMessage(error, &errorMsg) == ERROR_ok) {  /* Query printable error */
        printf("Error querying client ID: %s\n", errorMsg);
 ts3client_freeMemory(errorMsg);  /* Release memory only if function succeeded */
    }
}
```

In addition to actively querying errors like above, error codes can be sent by the server to the client. In that case the following event is called:

```
void onServerErrorEvent(serverConnectionHandlerID, errorMessage, error, returnCode,
extraMessage);

anyID serverConnectionHandlerID;
const char* errorMessage;
```

```
unsigned int error;
const char* returnCode;
const char* extraMessage;
```

Parameters

- *serverConnectionHandlerID*

  The connection handler ID of the server who sent the error event.

- *errorMessage*

  String containing a verbose error message, encoded in UTF-8 format.

- *error*

  Error code as defined in `public_errors.h`.

- *returnCode*

  String containing the return code if it has been set by the Client Lib function call which caused this error event.

  See return code documentation.

- *extraMessage*

  Can contain additional information about the occured error. If no additional information is available, this parameter is an empty string.

Logging

The TeamSpeak 3 Client Lib offers basic logging functions:

```
unsigned int ts3client_logMessage(logMessage, severity, channel, logID);

const char* logMessage;
LogLevel severity;
const char* channel;
anyID logID;
```

Parameters

- *logMessage*

  Text written to log.

- *severity*

  The level of the message, warning or error. Defined by the enum LogLevel in `clientlib_publicdefinitions.h`:

```
enum LogLevel {
    LogLevel_CRITICAL = 0, //these messages stop the program
    LogLevel_ERROR,        //everything that is really bad, but not so bad we need to shut down
    LogLevel_WARNING,      //everything that *might* be bad
    LogLevel_DEBUG,        //output that might help find a problem
    LogLevel_INFO,         //informational output, like "starting database version x.y.z"
    LogLevel_DEVEL         //developer only output (will not be displayed in release mode)
};
```

- *channel*

  Custom text to categorize the message channel (i.e. "Client", "Sound").

  Pass an empty string if unused.

- *logID*

  Server connection handler ID to identify the current server connection when using multiple connections.

  Pass 0 if unused.

All strings need to be encoded in UTF-8 format.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Log messages can be printed to stdout, logged to a file logs/ts3client_[date]__[time].log and sent to user-defined callbacks. The log output behaviour is defined when initialzing the client library with ts3client_initClientLib.

Unless user-defined logging is used, program execution will halt on a log message with severity *LogLevel_CRITICAL*.

User-defined logging

If user-defined logging was enabled when initialzing the Client Lib by passing *LogType_USERLOGGING* to the *usedLogTypes* parameter of ts3client_initClientLib, log messages will be sent to the following callback, which allows user customizable logging and handling or critical errors:

```
void onUserLoggingMessageEvent(logMessage, logLevel, logChannel, logID, logTime, completeLogString);
```

```
const char* logMessage;
int logLevel;
const char* logChannel;
anyID logID;
const char* logTime;
const char* completeLogString;
```

Most callback parameters reflect the arguments passed to the logMessage function.

Parameters

- *logMessage*

  Actual log message text.

- *logLevel*

  Severity of log message, defined by the enum LogLevel. Note that only log messages of a level higher than the one configured with `ts3client_setLogVerbosity` will appear.

- *logChannel*

  Optional custom text to categorize the message channel.

- *logID*

  Server connection handler ID identifying the current server connection when using multiple connections.

- *logTime*

  String with date and time when the log message occured.

- *completeLogString*

  Provides a verbose log message including all previous parameters for convinience.

The severity of log messages that are passed to above callback can be configured with:

```
unsigned int ts3client_setLogVerbosity(logVerbosity);

enum LogLevel logVerbosity;
```

Parameters

- *logVerbosity*

  Only messages with a log level equal or higher than *logVerbosity* will be sent to the callback. The default value is *LogLevel_DEVEL*.

  For example, after calling

  ```
  ts3client_setLogVerbosity(LogLevel_ERROR);
  ```

  only log messages of level *LogLevel_ERROR* and *LogLevel_CRITICAL* will be passed to `onUserLoggingMessageEvent`.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Using playback and capture modes and devices

The Client Lib takes care of initializing, using and releasing sound playback and capture devices. As a cross-platform library, the Client Lib handles all the low-level interfacing for multiple operating systems with Speex, FMOD, DirectSound, CoreAudio, ALSA and OSS.

All strings passed to and from the Client Lib have to be encoded in UTF-8 format.

Initializing modes and devices

To initialize a playback and capture device for a TeamSpeak 3 server connection handler, call

```
unsigned int ts3client_openPlaybackDevice(serverConnectionHandlerID, modeID, playbackDevice);

anyID serverConnectionHandlerID;
int modeID;
const char* playbackDevice;
```

Parameters

- *serverConnectionHandlerID*

  Connection handler of the server on which you want to initialize the playback device.

- *modeID*

  The playback mode to use. Valid modes are returned by `ts3client_getDefaultPlayBackMode` and `ts3client_getPlaybackModeList`.

- *playbackDevice*

  Valid parameters are:

  - The *device* parameter returned by `ts3client_getDefaultPlaybackDevice`
  - One of the *device* parameters returned by `ts3client_getPlaybackDeviceList`
  - Empty string or NULL to initialize the default playback device.
  - Linux with Alsa only: Custom device name in the form of: “hw:1,0”.

  The string needs to be encoded in UTF-8 format.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. A likely error is *ERROR_sound_could_not_open_playback_device* if FMOD fails to find a usable playback device.

```
unsigned int ts3client_openCaptureDevice(serverConnectionHandlerID, modeID, captureDevice);

anyID serverConnectionHandlerID;
int modeID;
const char* captureDevice;
```

Parameters

- *serverConnectionHandlerID*

Connection handler of the server on which you want to initialize the capture device.

- *modeID*

  The capture mode to use. Valid modes are returned by `ts3client_getDefaultCaptureMode` and `ts3client_getCaptureModeList`.

- *captureDevice*

  Valid parameters are:

  - The *device* parameter returned by `ts3client_getDefaultCaptureDevice`
  - One of the *device* parameters returned by `ts3client_getCaptureDeviceList`
  - Empty string or NULL to initialize the default capture device. Encoded in UTF-8 format.
  - Linux with Alsa only: Custom device name in the form of: “hw:1,0”.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. Likely errors are *ERROR_sound_could_not_open_capture_device* if the device fails to open or *ERROR_sound_handler_has_device* if the device is already opened. To avoid this problem, it is recommended to close the capture device before opening it again.

Querying available modes and devices

Various playback and capture modes are available, for example DirectSound on all Windows platforms or “Windows Audio Session API” for Windows Vista. It is important to note that the available devices depend on the current mode; not all devices are available for all modes.

The default playback and capture modes can be queried with:

```
unsigned int ts3client_getDefaultPlayBackMode(result);

int* result;



unsigned int ts3client_getDefaultCaptureMode(result);

int* result;
```

Parameters

- *result*

  Address of a variable that receives the default playback or capture mode. The value can be used as parameter for the functions querying and opening devices.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

All available playback and capture modes can be queried with:

```
unsigned int ts3client_getPlaybackModeList(result);

char**** result;
```

```
unsigned int ts3client_getCaptureModeList(result);

char**** result;
```

Parameters

- *result*

  Address of a variable that receives a NULL-terminated array { { char* modeID, char* name }, { char* modeID, char* name }, ... , NULL } listing available playback or capture modes.

  Unless the function returns an error, the caller must release modeID, name and the array itself using ts3client_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. In case of an error, the result array is uninitialized and must not be released.

Example to query all available playback devices:

```
char ***array;

if(ts3client_getPlaybackModeList(&array) == ERROR_ok) {
    for(int i=0; array[i] != NULL; ++i) {
        char *modeID = array[i][0];
 char *name = array[i][1];
 // ...
 ts3client_freeMemory(array[i][0]);
 ts3client_freeMemory(array[i][1]);
 ts3client_freeMemory(array[i]);
    }
    ts3client_freeMemory(array);
}
```

Playback and capture devices available for the given mode can be listed, as well as the current operating systems default. The returned values device can be used to initialize the devices.

To query the default playback and capture devices, call

```
unsigned int ts3client_getDefaultPlaybackDevice(modeID, result);

int modeID;
```

```
char*** result;
```

```
unsigned int ts3client_getDefaultCaptureDevice(modeID, result);

int modeID;
char*** result;
```

Parameters

- *modeID*

  Defines the playback/capture mode to use. For different modes there might be different default devices. Valid modes are returned by `ts3client_getDefaultPlayBackMode` / `ts3client_getDefaultCaptureMode` and `ts3client_getPlaybackModeList` / `ts3client_getCaptureModeList`.

- *result*

  Address of a variable that receives an array { char* name, char* device } (not NULL-terminated, always returns a tuple).

  Unless the function returns an error, the caller must free name, device and the array itself using `ts3client_freeMemory`.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. In case of an error, the result array is uninitialized and must not be released.

Example to query the default playback device:

```
int modeID;
char** defaultPlaybackDevice;

if(ts3client_getDefaultPlayBackMode(&modeID) != ERROR_ok) {
    printf("Error getting default playback mode\n");
    return;
}

if(ts3client_getDefaultPlaybackDevice(modeID, &defaultPlaybackDevice) != ERROR_ok) {
    printf("Error getting default playback device\n");
    return;
}

if(defaultPlaybackDevice != NULL) {
    char *name = defaultPlaybackDevice[0];
    char *device = defaultPlaybackDevice[1];
    // ...
    ts3client_freeMemory(defaultPlaybackDevice[0]);
    ts3client_freeMemory(defaultPlaybackDevice[1]);
    ts3client_freeMemory(defaultPlaybackDevice);
}
```

To get a list of all available playback and capture devices for the specified mode, call

```
unsigned int ts3client_getPlaybackDeviceList(modeID, result);
```

```
int modeID;
char**** result;
```

```
unsigned int ts3client_getCaptureDeviceList(modeID, result);

int modeID;
char**** result;
```

Parameters

- *modeID*

  Defines the playback/capture mode to use. For different modes there might be different device lists. Valid modes are returned by `ts3client_getDefaultPlayBackMode` / `ts3client_getDefaultCaptureMode` and `ts3client_getPlaybackModeList` / `ts3client_getCaptureModeList`.

- *result*

  Address of a variable that receives a NULL-terminated array { { char* name, char* device }, { char* name, char* device }, ... , NULL }.

  Unless the function returns an error, the array needs to be freed using `ts3client_freeMemory`.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. In case of an error, the result array is uninitialized and must not be released.

Example to query all available playback devices:

```
int modeID;
char*** array;

if(ts3client_getDefaultPlayBackMode(& modeID) != ERROR_ok) {
    printf("Error getting default playback mode\n");
    return;
}

if(ts3client_getPlaybackDeviceList(modeID, &array) != ERROR_ok) {
    printf("Error getting playback device list\n");
    return;
}

for(int i=0; array[i] != NULL; ++i) {
    char *name = array[i][0];
    char *device = array[i][1];
    // ...
    ts3client_freeMemory(array[i][0]);
    ts3client_freeMemory(array[i][1]);
    ts3client_freeMemory(array[i]);
}
ts3client_freeMemory(array);
```

The string *device* string can be used as parameter for `ts3client_openPlaybackDevice` / `ts3client_openCaptureDevice`.

Checking current modes and devices

The currently used playback and capture modes for a given server connection handler can be checked with:

```
unsigned int ts3client_getCurrentPlayBackMode(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
int* result;
```

```
unsigned int ts3client_getCurrentCaptureMode(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
int* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the current playback or capture modes are queried.

- *result*

  Address of a variable that receives the current playback or capture mode.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Check the currently used playback and capture devices for a given server connection handler with:

```
unsigned int ts3client_getCurrentPlaybackDeviceName(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
char** result;
```

```
unsigned int ts3client_getCurrentCaptureDeviceName(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
char** result;
```

Parameters

- *serverConnectionHandlerID*

ID of the server connection handler for which the current playback or capture devices are queried.

- *result*

  Address of a variable that receives the current playback or capture device. Unless the function returns an error, the string must be released using ts3client_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result string is uninitialized and must not be released.

Closing devices

To close the capture and playback devices for a given server connection handler:

```
unsigned int ts3client_closeCaptureDevice(serverConnectionHandlerID);

anyID serverConnectionHandlerID;


unsigned int ts3client_closePlaybackDevice(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the playback or capture device should be closed.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

ts3client_closePlaybackDevice will not block until all current sounds have finished playing but will shutdown the device immediately, possibly interrupting the still playing sounds. To shutdown the playback device more gracefully, use the following function:

```
unsigned int ts3client_initiateGracefulPlaybackShutdown(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the playback or capture device should be shut down.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

While ts3client_initiateGracefulPlaybackShutdown will not block until all sounds have finished playing, too, it will notify the client when the playback device can be safely closed by sending the callback:

```
void onPlaybackShutdownCompleteEvent(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the playback device has been shut down.

Example code to gracefully shutdown the playback devicef:

```
/* Instead of calling ts3client_closePlaybackDevice() directly */
if(ts3client_initiateGracefulPlaybackShutdown(currentScHandlerID) != ERROR_ok) {
    printf("Failed to initiate graceful playback shutdown\n");
    return;
}

/* Event notifying the playback device has been shutdown */
void my_onPlaybackShutdownCompleteEvent(anyID scHandlerID) {
    /* Now we can safely close the device */
    if(ts3client_closePlaybackDevice(scHandlerID) != ERROR_ok) {
        printf("Error closing playback device\n");
    }
}
```

Note

Devices are closed automatically when calling ts3client_destroyServerConnectionHandler.

Note

To change a device, close it first and then reopen it.

Activating the capture device

Note

Using this function is only required when connecting to multiple servers.

When connecting to multiple servers with the same client, the capture device can only be active for one server at the same time. As soon as the client connects to a new server, the Client Lib will deactivate the capture device of the previously active server. When a user wants to talk to that previous server again, the client needs to reactivate the capture device.

```
unsigned int ts3client_activateCaptureDevice(serverConnectionHandlerID);
```

```
anyID serverConnectionHandlerID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the capture device should be activated.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

If the capture device is already active, this function has no effect.

Opening a new capture device will automatically activate it, so calling this function is only necessary with multiple server connections and when reactivating a previously deactivated device.

If the capture device for a given server connection handler has been deactivated by the Client Lib, the flag *CLIENT_INPUT_HARDWARE* will be set. This can be queried with the function ts3client_getClientSelfVariableAsInt.

Custom FMOD objects

Using custom FMOD system objects

Instead of using the standard Client Lib functions ts3client_openPlaybackDevice, ts3client_openCaptureDevice and ts3client_activateCaptureDevice it is possible to supply a *FMOD::System* object of your own that is then used by the TeamSpeak 3 SDK. This allows you to integrate your existing FMOD solution for your application with the FMOD [http://www.fmod.org] solution used by TeamSpeak 3, or even if you don't use FMOD in your application you can use this functionality if you want more control over the way the *FMOD::System* objects are used.

By managing the *FMOD::System* objects yourself and by using the later described onFMODChannelCreatedEvent callback you will be able to use advanced FMOD functionality that the TeamSpeak 3 client SDK does not use and does not expose to be used through the clientlib.h API, for example custom DSP effects or modelling the 3D world.

Note

Using custom FMOD objects is entirely optional.

The function to open a playback device using a custom *FMOD::System* object is an alternative to ts3client_openPlaybackDevice:

```
unsigned int ts3client_openCustomPlaybackDevice(serverConnectionHandlerID, fmodSystem);
```

```
anyID serverConnectionHandlerID;
void* fmodSystem;
```

The function to open a capture device using a custom *FMOD::System* object is an alternative to ts3client_openCaptureDevice:

```
unsigned int ts3client_openCustomCaptureDevice(serverConnectionHandlerID, fmodSys-
tem, fmodDriverID);

anyID serverConnectionHandlerID;
void* fmodSystem;
int fmodDriverID;
```

The function to activate the capture device using a custom *FMOD::System* object is an alternative to ts3client_activateCaptureDevice:

```
unsigned int ts3client_activateCustomCaptureDevice(serverConnectionHandlerID,
fmodSystem, fmodDriverID);

anyID serverConnectionHandlerID;
void* fmodSystem;
int fmodDriverID;
```

Parameters

- *serverConnectionHandlerID*

  Connection handler of the server on which you want to initialize or activate the device.

- *fmodSystem*

  A *FMOD::System* object created and customized by your code, as shown in the following examples.

  If you use the FMOD C API, you can pass a *FMOD_SYSTEM** object instead of *FMOD::System**, as the two are interchangable.

- *fmodDriverID*

  Enumerated FMOD driver ID, see the FMOD documentation for details.

All functions return *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Note

The TeamSpeak 3 SDK uses *void** pointers to avoid the dependency on FMOD, as custom FMOD objects is an optional feature.

Simple example how to create a *FMOD::System* object for opening a capture device:

```
FMOD::System* pFMODSystem = NULL;
FMOD_RESULT result;

result = FMOD::System_Create(&pFMODSystem);
```

```
if(result != FMOD_OK) {
    cout << "System_Create: " << FMOD_ErrorString(result) << endl;
    return;
}

result= pFMODSystem->setOutput(static_cast<FMOD_OUTPUTTYPE>(modeID));
if(result != FMOD_OK) {
    cout << "setOutput failed (modeID: " << modeID << "): " << FMOD_ErrorString(result) << endl;
    return;
}

result = pFMODSystem->init(32, FMOD_INIT_NORMAL, NULL);
if(result != FMOD_OK) {
    cout << "init: " << FMOD_ErrorString(result) << endl;
    return;
}
```

Simple example how to create a *FMOD::System* object for opening a playback device:

```
FMOD::System* system = NULL;
FMOD_RESULT result;

result = FMOD::System_Create(&system);
if(result != FMOD_OK) {
    cout << "System_Create: " << FMOD_ErrorString(result) << endl;
    return false;
}

result= system->setOutput(static_cast<FMOD_OUTPUTTYPE>(modeID));
if(result != FMOD_OK) {
    cout << "setOutput failed (modeID: " << modeID << "): " << FMOD_ErrorString(result) << endl;
    return false;
}

result= system->setDriver(driverID);
if(result != FMOD_OK) {
    cout << "setDriver: " << FMOD_ErrorString(result) << endl;
    return false;
}

result = system->init(1000, FMOD_INIT_NORMAL, NULL);
if(result != FMOD_OK) {
    cout << "init: " << FMOD_ErrorString(result) << endl;
    return false;
}
```

See FMOD Ex Frequently Asked Questions [http://www.fmod.org/forum/viewtopic.php?t=9305] for the recommended way to initialize the output device and the FMOD [http://www.fmod.org] API documentation for further details.

When the TeamSpeak 3 SDK is using a passed-in *FMOD::System* object, it will not call `close()` or `release()` on the *FMOD::System* object when finished with it, but rather notify the SDK user via a callback that the SDK has finished using the object. You can either call `release()` on the *FMOD::System* object or - if you are still using it in your own application - do nothing.

When closing a capture device, which has been previously opened using `ts3client_openCustomCaptureDevice`, the following callback allows to clean-up the custom *FMOD::System* object if desired:

`void` **`onCustomCaptureDeviceCloseEvent`**`(`*`serverConnectionHandlerID, fmodSystem`*`);`

`anyID` *`serverConnectionHandlerID`*`;`

```
void* fmodSystem;
```

When closing a playback device, which has been previously opened using `ts3client_openCustomPlaybackDevice`, the following callback allows to clean-up the custom *`FMOD::System`* object if desired:

```
void onCustomPlaybackDeviceCloseEvent(serverConnectionHandlerID, fmodSystem);

anyID serverConnectionHandlerID;
void* fmodSystem;
```

Note

Note that you still use the standard ts3client_closePlaybackDevice and ts3client_closeCaptureDevice function to close devices opened using `ts3client_openCustomCaptureDevice` and `ts3client_openCustomPlaybackDevice`.

Customizing FMOD channel objects

FMOD channels are created by the Client Lib whenever a user starts talking. By implementing the following callback, additional settings can be applied to the *`FMOD::Channel`* object.

Note

The `onFMODChannelCreatedEvent` callback is always called, no matter if devices were opened with the standard or custom functions. This allows to apply additional settings to FMOD Channels even if the custom FMOD System objects mechanism is not used.

If you do not need this callback, just set the function pointer to *`NULL`* in the ClientUIFunctions parameter for `ts3client_initClientLib`.

The order of operations when a client starts talking is:

- FMOD channel is created in a paused state.
- Client Lib sets standard options on the *`FMOD::Channel`* object, like 3D sound settings as set with `ts3client_fmod_Channelset3DAttributes`.
- `onFMODChannelCreatedEvent` is called and allows you to apply additional settings on the *`FMOD::Channel`* object, or to even overwrite the default settings that were made in the previous step.
- The FMOD channel is unpaused and the client starts talking.

```
void onFMODChannelCreatedEvent(serverConnectionHandlerID, clientID, fmodChannel);

anyID serverConnectionHandlerID;
anyID clientID;
void* fmodChannel;
```

Parameters

- *serverConnectionHandlerID*

  Connection handler of the server on which the FMOD channel is created.

- *clientID*

  ID of the client who starts talking.

- *fmodChannel*

  A *FMOD::Channel* object on which additional settings can be applied.

  If you use the FMOD C API, cast the *void* fmodChannel* object to *FMOD_CHANNEL** instead of *FMOD::Channel**, as the two are interchangable.

The callback should return quickly to avoid delaying unpausing the FMOD channel.

Querying the current FMOD System objects

The currently used FMOD System playback and capture objects - may it be a standard or custom object - can be queried from the clientlib if you want to use them for own FMOD operations like for example playing wave files.

Request the current playback device with:

```
unsigned int ts3client_getCurrentPlaybackDevice(serverConnectionHandlerID, fmodSys-
temResult);

anyID serverConnectionHandlerID;
void** fmodSystemResult;
```

Request the current capture device with:

```
unsigned int ts3client_getCurrentCaptureDevice(serverConnectionHandlerID, fmodSys-
temResult);

anyID serverConnectionHandlerID;
void** fmodSystemResult;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the FMOD system devices should be queried.

- *serverConnectionHandlerID*

  Address of a variable that receives the requested FMOD System object.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If the function fails, *fmodSystemResult* is undefined.

Sound codecs

TeamSpeak 3 supports three different sound sampling rates:

- Speex Narrowband (8 kHz)
- Speex Wideband (16 kHz)
- Speex Ultra-Wideband (32 kHz)

Bandwidth usage generally depends on the encoders quality setting.

Quality	Narrowband bitrate (bps)	Wideband bitrate (bps)	Ultra-Wideband bitrate (bps)
0	2,150	3,950	5,750
1	3,950	5,750	7,550
2	5,950	7,750	9,550
3	8,000	9,800	11,600
4	8,000	12,800	14,600
5	11,000	16,800	18,600
6	11,000	20,600	22,400
7	15,000	23,800	25,600
8	15,000	27,800	29,600
9	18,200	34,400	36,200
10	24,600	42,400	44,200

The availability of the 8 kHz narrowband codec should cater for the needs of low-bandwidth users at the cost of overall sound quality.

Users need to use the same codec when talking to each others. The smallest unit of participants using the same codec is a channel. Different channels on the same TeamSpeak 3 server can use different codecs. The channel codec should be customizable by the users to allow for flexibility concerning bandwidth vs. quality concerns.

The codec can be set or changed for a given channel using the function ts3client_setChannelVariableAsInt by passing *CHANNEL_CODEC* for the properties flag:

```
ts3client_setChannelVariableAsInt(scHandlerID, channelID, CHANNEL_CODEC, codec);
```

For the argument *codec* pass a value of 0 for Narrowband (8 kHz), 1 for Wideband (16 kHz) and 2 for Ultra-Wideband (32 kHz).

For details on using the function ts3client_setChannelVariableAsInt see the appropriate section on changing channel data.

Encoder options

Speech quality and bandwidth usage depend on the used Speex encoder. As Speex is a lossy code, the quality value controls the balance between voice quality and network traffic. Valid quality values range from 0 to 10, default is 7. The encoding quality can be configured for each channel using the *CHANNEL_CODEC_QUALITY* property. The currently used channel codec, codec quality and estimated average used bitrate (without overhead) can be queried with ts3client_getEncodeConfigValue.

Note

Encoder options are tied to a capture device, so querying the values only makes sense after a device has been opened.

All strings passed from the Client Lib are encoded in UTF-8 format.

```
unsigned int ts3client_getEncodeConfigValue(serverConnectionHandlerID, ident, result);

anyID serverConnectionHandlerID;
const char* ident;
char** result;
```

- *serverConnectionHandlerID*

  Server connection handler ID

- *ident*

  String containing the queried encoder option. Available values are “name”, “quality” and “bitrate”.

- *result*

  Address of a variable that receives the result string. Unless an error occured, the result string must be released using ts3client_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result string is uninitialized and must not be released.

To adjust the channel codec quality to a value of 5, you would call:

```
ts3client_setChannelVariableAsInt(scHandlerID, channelID, CHANNEL_CODEC_QUALITY, 5);
```

See the chapter about channel information for details about how to set channel variables.

To query information about the current channel quality, do:

```
char *name, *quality, *bitrate;
ts3client_getEncodeConfigValue(scHandlerID, "name", &name);
ts3client_getEncodeConfigValue(scHandlerID, "quality", &quality);
ts3client_getEncodeConfigValue(scHandlerID, "bitrate", &bitrate);

printf("Name = %s, quality = %s, bitrate = %s\n", name, quality, bitrate);

ts3client_freeMemory(name);
ts3client_freeMemory(quality);
```

```
ts3client_freeMemory(bitrate);
```

Preprocessor options

Sound input is preprocessed by the Client Lib before the data is encoded and sent to the TeamSpeak 3 server. The preprocessor is responsible for noise suppression, automatic gain control (AGC), voice activity detection (VAD) and echo canceling.

The preprocessor can be controlled by setting various preprocessor flags. These flags are unique to each server connection.

Note

Preprocessor flags are tied to a capture device, so changing the values only makes sense after a device has been opened.

Preprocessor flags can be queried using

```
unsigned int ts3client_getPreProcessorConfigValue(serverConnectionHandlerID, ident,
result);
```

```
anyID serverConnectionHandlerID;
const char* ident;
char** result;
```

Parameters

- *serverConnectionHandlerID*

  The server connection handler ID.

- *ident*

  The proprocessor flag to be queried. The following keys are available:

  - “name”

    Type of the used preprocessor. Currently this returns a constant string “Speex preprocessor”.

  - “denoise”

    Check if noise suppression is enabled. Returns “true” or “false”.

  - “vad”

    Check if Voice Activity Detection is enabled. Returns “true” or “false”.

  - “voiceactivation_level”

    Checks the Voice Activity Detection level in decibel. Returns a string with a numeric value, convert this to an integer.

  - “vad_extrabuffersize”

    Checks Voice Activity Detection extrabuffer size. Returns a string with a numeric value.

- “agc”

  Check if Automatic Gain Control is enabled. Returns “true” or “false”.

- “agc_level”

  Checks AGC level. Returns a string with a numeric value.

- “agc_max_gain”

  Checks AGC max gain. Returns a string with a numeric value.

- “echo_canceling”

  Check if echo canceling is enabled. Returns “true” or “false”.

- “is_playback_echo_canceled”

  Query a server connection handler to see if its playback device is being echo canceled.

- “canceled_amount”

  Returns a number indicating how much echo is being removed by the echo canceler, if enabled for the playback device of the specified server connection handler.

- *result*

  Address of a variable that receives the result as a string encoded in UTF-8 format. If no error occured the returned string must be released using `ts3client_freeMemory`.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. If an error has occured, the result string is uninitialized and must not be released.

To configure the proprocessor use

```
unsigned int ts3client_setPreProcessorConfigValue(serverConnectionHandlerID, ident, value);
```

```
anyID serverConnectionHandlerID;
const char* ident;
const char* value;
```

Parameters

- *serverConnectionHandlerID*

  The server connection handler ID.

- *ident*

  The preprocessor flag to be configure. The following keys can be changed:

- “denoise”

  Enable or disable noise suppression. Value can be “true” or “false”. Enabled by default.

- “vad”

  Enable or disable Voice Activity Detection. Value can be “true” or “false”. Enabled by default.

- “voiceactivation_level”

  Voice Activity Detection level in decibel. Numeric value converted to string. A high voice activation level means you have to speak louder into the microphone in order to start transmitting.

  Reasonable values range from -50 to 50. Default is 0.

  To adjust the VAD level in your client, you can call `ts3client_getPreProcessorInfoValueFloat` with the identifier “decibel_last_period” over a period of time to query the current voice input level.

- “vad_extrabuffersize”

  Voice Activity Detection extrabuffer size. Numeric value converted to string. Should be “0” to “8”, defaults to “2”. Lower value means faster transmission, higher value means better VAD quality but higher latency.

- “agc”

  Enable or disable Automatic Gain Control. Value can be “true” or “false”. Enabled by default.

- “agc_level”

  AGC level. Numeric value converted to string. Default is “16000”.

- “agc_max_gain”

  AGC max gain. Numeric value converted to string. Default is “30”.

- “echo_canceling”

  Enable or disable echo canceling. Value can be “true” or “false”. Disabled by default.

- “set_echo_cancelers_playback”

  Set the playback device of the selected server connection handler to be the device used for echo canceling. Pass an empty string as *`value`* to `ts3client_setPreProcessorConfigValue`, the playback device is automatically selected by the specified server connection handler.

- *`value`*

  String value to be set for the given preprocessor identifier. In case of on/off switches, use “true” or “false”.

Returns *`ERROR_ok`* on success, otherwise an error code as defined in `public_errors.h`.

Note

It is not necessary to change all those values. The default values are reasonable. “voiceactivation_level” is often the only value that needs to be adjusted.

The following function retrieves preprocessor information as a floating-point variable instead of a string:

```
unsigned int ts3client_getPreProcessorInfoValueFloat(serverConnectionHandlerID,
ident, result);

anyID serverConnectionHandlerID;
const char* ident;
float* result;
```

Parameters

- *serverConnectionHandlerID*

  The server connection handler ID.

- *ident*

  The proprocessor flag to be queried. Currently the only valid identifier for this function is “decibel_last_period”, which can be used to adjust the VAD level as described above.

- *result*

  Address of a variable that receives the result value as a float.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Playback options

Sound output can be configured using playback options. Currently the output value can be adjusted.

Playback options can be queried:

```
unsigned int ts3client_getPlaybackConfigValueAsFloat(serverConnectionHandlerID,
ident, result);

anyID serverConnectionHandlerID;
const char* ident;
float* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the playback option is queried.

- *ident*

Identifier of the parameter to be configured. Possible values are:

- "volume_modifier"

  Modify the voice volume of other speakers. Value is in decibel, so 0 is no modification, negative values make the signal quieter and values greater than zero boost the signal louder than it is. Be careful with high positive values, as you can really cause bad audio quality due to clipping. The maximum possible Value is 30.

  Zero and all negative values cannot cause clipping and distortion, and are preferred for optimal audio quality. Values greater than zero and less than +6 dB might cause moderate clipping and distortion, but should still be within acceptable bounds. Values greater than +6 dB will cause clipping and distortion that will negatively affect your audio quality. It is advised to choose lower values. Generally we recommend to not allow values higher than 15 db.

- *result*

  Address of a variable that receives the playback configuration value as floating-point number.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

To change playback options, call:

```
unsigned int ts3client_setPlaybackConfigValue(serverConnectionHandlerID, ident, value);

anyID serverConnectionHandlerID;
const char* ident;
const char* value;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the playback option is queried.

- *ident*

  Identifier of the parameter to be configured. The values are the same as in ts3client_getPlaybackConfigValueAsFloat above.

- *value*

  String with the value to set the option to, encoded in UTF-8 format.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Note

Playback options are tied to a playback device, so changing the values only makes sense after a device has been opened.

Example code:

```
unsigned int error;
float value;

if((error = ts3client_setPlaybackConfigValue(scHandlerID, "volume_modifier", "5.5")) != ERROR_ok) {
    printf("Error setting playback config value: %d\n", error);
    return;
}

if((error = ts3client_getPlaybackConfigValueAsFloat(scHandlerID, "volume_modifier", &value)) != ERROR_ok) {
    printf("Error getting playback config value: %d\n", error);
    return;
}

printf("Volume modifier playback option: %f\n", value);
```

In addition to changing the global voice volume modifier of all speakers by changing the “volume_modifier” parameter, voice volume of individual clients can be adjusted with:

```
unsigned int ts3client_setClientVolumeModifier(serverConnectionHandlerID, clientID, value);

anyID serverConnectionHandlerID;
anyID clientID;
float value;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the client volume modifier should be adjusted.

- *clientID*

  ID of the client whose volume modifier should be adjusted.

- *value*

  The new client volume modifier value as float.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

When calculating the volume for individual clients, both the global and client volume modifiers will be taken into account.

Client volume modifiers are valid as long as the specified client is visible. Once the client leaves visibility by joining an unsubscribed channel or disconnecting from the server, the client volume modifier will be lost. When the client enters visibility again, the modifier has to be set again by calling this function.

Example:/

```
unsigned int error;
```

```
anyID clientID = 123;
float value = 10.0f;

if((error = ts3client_setClientVolumeModifier(scHandlerID, clientID, value)) != ERROR_ok) {
    printf("Error setting client volume modifier: %d\n", error);
 return;
}
```

3D Sound

TeamSpeak 3 supports 3D sound to assign each speaker a unique position in 3D space. Provided are wrapper functions to the FMOD 3D sound system to modify the 3D position, velocity and orientation of own and foreign clients.

Generally the struct TS3CLIENT_FMOD_VECTOR describes a vector in 3D space:

```
typedef struct {
    float x;        /* X coordinate in 3D space. */
    float y;        /* Y coordinate in 3D space. */
    float z;        /* Z coordinate in 3D space. */
} TS3CLIENT_FMOD_VECTOR;
```

To set the position, velocity and orientation of the own client in 3D space, call:

```
unsigned int ts3client_fmod_Systemset3DListenerAttributes(serverConnectionHandlerID, position, velocity, forward, up);

anyID serverConnectionHandlerID;
const TS3_FMOD_VECTOR* position;
const TS3_FMOD_VECTOR* velocity;
const TS3_FMOD_VECTOR* forward;
const TS3_FMOD_VECTOR* up;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the 3D sound listener attributes are to be set.

- *position*

  3D position of the own client.

  If passing NULL, the parameter is ignored and the value not updated.

- *velocity*

  Velocity of the own client in “distance units per second”. A “distance unit” is specified by the function `ts3client_fmod_Systemset3DSettings`, the default is a scale of 1.0 describing one meter.

  If passing NULL, the parameter is ignored and the value not updated.

- *forward*

Forward orientation of the listener. The vector must be of unit length and perpendicular to the up vector.

If passing NULL, the parameter is ignored and the value not updated.

- *up*

Upward orientation of the listener. The vector must be of unit length and perpendicular to the forward vector.

If passing NULL, the parameter is ignored and the value not updated.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

This function is a wrapper for the FMOD function System::set3DListenerAttributes. Please see the FMOD documentation for details.

To adjust FMOD 3D sound system settings use:

```
unsigned int ts3client_fmod_Systemset3DSettings(serverConnectionHandlerID, dopplerScale, distanceFactor, rolloffScale);

anyID serverConnectionHandlerID;
float dopplerScale;
float distanceFactor;
float rolloffScale;
```

Parameters

- *serverConnectionHandlerID*

ID of the server connection handler on which the 3D sound system settings are to be adjusted.

- *dopplerScale*

Scaling factor for doppler shift, default is 1.0

- *distanceFactor*

Relative distance factor. Default is 1.0 = 1 meter

- *rolloffScale*

Scaling factor for 3D sound rolloff.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

This function is a wrapper for the FMOD function System::set3DSettings. Please see the FMOD documentation for details.

To adjust a clients position and velocity in 3D space, call:

```
unsigned int ts3client_fmod_Channelset3DAttributes(serverConnectionHandlerID, clientID, position, velocity);

anyID serverConnectionHandlerID;
anyID clientID;
const TS3_FMOD_VECTOR* position;
const TS3_FMOD_VECTOR* velocity;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the 3D sound channel attributes are to be adjusted.

- *clientID*

  ID of the client to adjust.

- *position*

  Vector specifying the position of the given client in 3D space.

  When passing NULL, the parameter will be ignored.

- *velocity*

  Vector describing clients velocity in “distance units per second” in 3D space. A “distance unit” is specified by the function `ts3client_fmod_Systemset3DSettings`, the default is a scale of 1.0 describing one meter.

  When passing NULL, the parameter will be ignored.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

This function is a wrapper to the FMOD function `Channel::set3DAttributes`. See the FMOD documentation for details.

Query available servers, channels and clients

A client can connect to multiple servers. To list all currently existing server connection handlers, call:

```
anyID* ts3client_getServerConnectionHandlerList(result);

anyID** result;
```

Parameters

- *result*

Address of a variable that receives a NULL-termianted array of all currently existing server connection handler IDs. Unless an error occurs, the array must be released using `ts3client_freeMemory`.

Returns *`ERROR_ok`* on success, otherwise an error code as defined in `public_errors.h`. If an error has occured, the result array is uninitialized and must not be released.

A list of all channels on the specified virtual server can be queried with:

```
unsigned int ts3client_getChannelList(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
anyID** result;
```

Parameters

- *`serverConnectionHandlerID`*

  ID of the server connection handler for which the list of channels is requested.

- *`result`*

  Address of a variable that receives a NULL-termianted array of channel IDs. Unless an error occurs, the array must be released using `ts3client_freeMemory`.

Returns *`ERROR_ok`* on success, otherwise an error code as defined in `public_errors.h`. If an error has occured, the result array is uninitialized and must not be released.

To get a list of all currently visible clients on the specified virtual server:

```
unsigned intts3client_getClientList(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
anyID** result;
```

Parameters

- *`serverConnectionHandlerID`*

  ID of the server connection handler for which the list of clients is requested.

- *`result`*

  Address of a variable that receives a NULL-termianted array of client IDs. Unless an error occurs, the array must be released using `ts3client_freeMemory`.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result array is uninitialized and must not be released.

To get a list of all clients in the specified channel:

```
unsigned int  ts3client_getChannelClientList(serverConnectionHandlerID, channelID,
result);

anyID serverConnectionHandlerID;
anyID channelID;
anyID** result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the list of clients within the given channel is requested.

- *channelID*

  ID of the channel whose client list is requested.

- *result*

  Address of a variable that receives a NULL-termianted array of client IDs. Unless an error occurs, the array must be released using ts3client_freeMemory.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h. If an error has occured, the result array is uninitialized and must not be released.

To query the channel ID the specified client has currently joined:

```
unsigned int ts3client_getChannelOfClient(scHandlerID, clientID, result);

anyID scHandlerID;
anyID clientID;
anyID* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the channel ID is requested.

- *clientID*

ID of the client whose channel ID is requested.

- *result*

  Address of a variable that receives the ID of the channel the specified client has currently joined.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

To get the parent channel of a given channel:

```
unsigned int ts3client_getParentChannelOfChannel(scHandlerID, channelID, result);

anyID scHandlerID;
anyID channelID;
anyID* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the parent channel of the specified channel is requested.

- *channelID*

  ID of the channel whose parent channel ID is requested.

- *result*

  Address of a variable that receives the ID of the parent channel of the specified channel.

  If the specified channel has no parent channel, *result* will be set to the reserved channel ID 0.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Example code to print a list of all channels on a virtual server:

```
anyID* channels;

if(ts3client_getChannelList(serverID, &channels) == ERROR_ok) {
    for(int i=0; channels[i] != NULL; i++) {
        printf("Channel ID: %u\n", channels[i]);
    }
    ts3client_freeMemory(channels);
}
```

To print all visible clients:

```
anyID* clients;

if(ts3client_getClientList(scHandlerID, &clients) == ERROR_ok) {
```

```
    for(int i=0; clients[i] != NULL; i++) {
        printf("Client ID: %u\n", clients[i]);
    }
    ts3client_freeMemory(clients);
}
```

Example to print all clients who are member of channel with ID 123:

```
anyID channelID = 123;  /* Channel ID in this example */
anyID *clients;

if(ts3client_getChannelClientList(scHandlerID, channelID) == ERROR_ok) {
    for(int i=0; clients[i] != NULL; i++) {
        printf("Client ID: %u\n", clients[i]);
    }
    ts3client_freeMemory(clients);
}
```

Retrieve and store information

The Client Lib remembers a lot of information which have been passed through previously. The data is available to be queried by a client for convinience, so the interface code doesn't need to store the same information as well. The client can in many cases also modify the stored information for further processing by the server.

All strings passed to and from the Client Lib need to be encoded in UTF-8 format.

Client information

Information related to own client

Once connection to a TeamSpeak 3 server has been established, a unique client ID is assigned by the server. This ID can be queried with

```
unsigned int ts3client_getClientID(serverConnectionHandlerID, result);

anyID serverConnectionHandlerID;
anyID* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which we are querying the own client ID.

- *result*

  Address of a variable that receives the client ID. Client IDs start with the value 1.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Various information related about the own client can be checked with:

```
unsigned int ts3client_getClientSelfVariableAsInt(serverConnectionHandlerID, flag,
result);

anyID serverConnectionHandlerID;
ClientProperties flag;
int* result;
```

```
unsigned int ts3client_getClientSelfVariableAsString(serverConnectionHandlerID,
flag, result);

anyID serverConnectionHandlerID;
ClientProperties flag;
char** result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the information for the own client is requested.

- *flag*

  Client propery to query, see below.

- *result*

  Address of a variable which receives the result value as int or string, depending on which function is used. In case of a string, memory must be released using `ts3client_freeMemory`, unless an error occured.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. For the string version: If an error has occured, the result string is uninitialized and must not be released.

The parameter *flag* specifies the type of queried information. It is defined by the enum ClientProperties:

```
enum ClientProperties {
  CLIENT_UNIQUE_IDENTIFIER = 0,   //automatically up-to-date for any client "in view", can be used
                                  //to identify this particular client installation
  CLIENT_NICKNAME,                //automatically up-to-date for any client "in view"
  CLIENT_VERSION,                 //for other clients than ourself, this needs to be requested
                                  //(=> requestClientVariables)
  CLIENT_PLATFORM,                //for other clients than ourself, this needs to be requested
                                  //(=> requestClientVariables)
  CLIENT_FLAG_TALKING,            //automatically up-to-date for any client that can be heard
                                  //(in room / whisper)
  CLIENT_INPUT_MUTED,             //automatically up-to-date for any client "in view", this clients
                                  //microphone mute status
  CLIENT_OUTPUT_MUTED,            //automatically up-to-date for any client "in view", this clients
                                  //headphones/speakers mute status
  CLIENT_INPUT_HARDWARE,          //automatically up-to-date for any client "in view", this clients
                                  //microphone hardware status (is the capture device opened?)
  CLIENT_OUTPUT_HARDWARE,         //automatically up-to-date for any client "in view", this clients
                                  //headphone/speakers hardware status (is the playback device opened?)
  CLIENT_INPUT_DEACTIVATED,       //only usable for ourself, not propagated to the network
```

```
  CLIENT_IDLE_TIME,               //internal use
  CLIENT_DEFAULT_CHANNEL,         //only usable for ourself, the default channel we used to connect
                                  //on our last connection attempt
  CLIENT_DEFAULT_CHANNEL_PASSWORD,//internal use
  CLIENT_SERVER_PASSWORD,         //internal use
  CLIENT_META_DATA,               //automatically up-to-date for any client "in view", not used by
                                  //TeamSpeak, free storage for sdk users
  CLIENT_IS_MUTED,                //only make sense on the client side locally, "1" if this client is
                                  //currently muted by us, "0" if he is not
  CLIENT_IS_RECORDING,            //automatically up-to-date for any client "in view"
  CLIENT_VOLUME_MODIFICATOR,      //internal use
  CLIENT_ENDMARKER,
};
```

- *CLIENT_UNIQUE_IDENTIFIER*

  String: Unique ID for this client. Stays the same after restarting the application, so you can use this to identify individual user.

- *CLIENT_NICKNAME*

  Nickname used by the client. This value is always automatically updated for visible clients.

- *CLIENT_VERSION*

  Application version used by this client. Needs to be requested with `ts3client_requestClientVariables` unless called on own client.

- *CLIENT_PLATFORM*

  Operating system used by this client. Needs to be requested with `ts3client_requestClientVariables` unless called on own client.

- *CLIENT_FLAG_TALKING*

  Set when the client is currently sending voice data to the server. Always available for visible clients.

- *CLIENT_INPUT_MUTED*

  Indicates the mute status of the clients capture device. Possible values are defined by the enum MuteInputStatus. Always available for visible clients.

- *CLIENT_OUTPUT_MUTED*

  Indicates the mute status of the clients playback device. Possible values are defined by the enum MuteOutputStatus. Always available for visible clients.

- *CLIENT_INPUT_HARDWARE*

  Set if the clients capture device is not available. Possible values are defined by the enum HardwareInputStatus. Always available for visible clients.

- *CLIENT_OUTPUT_HARDWARE*

  Set if the clients playback device is not available. Possible values are defined by the enum HardwareOutputStatus. Always available for visible clients.

- *CLIENT_INPUT_DEACTIVATED*

Set when the capture device has been deactivated as used in Push-To-Talk. Possible values are defined by the enum InputDeactivationStatus. Only used for the own clients and not available for other clients as it doesn't get propagated to the server.

- *CLIENT_IDLE_TIME*

  Time the client has been idle. Needs to be requested with `ts3client_requestClientVariables`.

- *CLIENT_DEFAULT_CHANNEL*

  CLIENT_DEFAULT_CHANNEL_PASSWORD

  Default channel name and password used in the last `ts3client_startConnection` call. Only available for own client.

- *CLIENT_META_DATA*

  Not used by TeamSpeak 3, offers free storage for SDK users. Always available for visible clients.

- *CLIENT_IS_MUTED*

  Indicates a client has been locally muted with `ts3client_requestMuteClients`. Client-side only.

- *CLIENT_IS_RECORDING*

  Indicates a client is currently recording all voice data in his channel.

- *CLIENT_VOLUME_MODIFICATOR*

  The client volume modifier set by `ts3client_setClientVolumeModifier`.

Generally all types of information can be retrieved as both string or integer. However, in most cases the expected data type is obvious, like querying *CLIENT_NICKNAME* will clearly require to store the result as string.

Example 1: Query client nickname

```
char* nickname;

if(ts3client_getClientSelfVariableAsString(scHandlerID, CLIENT_NICKNAME, &nickname) == ERROR_ok) {
    printf("My nickname is: %s\n", s);
    ts3client_freeMemory(s);
}
```

Example 2: Check if own client is currently talking (to be exact: sending voice data)

```
int talking;

if(ts3client_getClientSelfVariableAsInt(scHandlerID, CLIENT_FLAG_TALKING, &talking) == ERROR_ok) {
    switch(talking) {
        case STATUS_TALKING:
            // I am currently talking
     break;
 case STATUS_NOT_TALKING:
     // I am currently not talking
     break;
 case STATUS_TALKING_WHILE_DISABLED:
     // I am talking while microphone is disabled
     break;
 default:
     printf("Invalid value for CLIENT_FLAG_TALKING\n");
```

```
    }
}
```

Information related to the own client can be modified with

```
unsigned int ts3client_setClientSelfVariableAsInt(serverConnectionHandlerID, flag, value);

anyID serverConnectionHandlerID;
ClientProperties flag;
int value;
```

```
unsigned int ts3client_setClientSelfVariableAsString(serverConnectionHandlerID, flag, value);

anyID serverConnectionHandlerID;
ClientProperties flag;
const char* value;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the information for the own client is changed.

- *flag*

  Client propery to query, see above.

- *value*

  Value the client property should be changed to.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Important

After modifying one or more client variables, you *must* flush the changes. Flushing ensures the changes are sent to the TeamSpeak 3 server.

```
unsigned int ts3client_flushClientSelfUpdates(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

For example, to change the own nickname:

```
/* Modify data */
if(ts3client_setClientSelfVariableAsString(scHandlerID, CLIENT_NICKNAME, "Joe") != ERROR_ok) {
```

```
    printf("Error setting client variable\n");
    return;
}

/* Flush changes */
if(ts3client_flushClientSelfUpdates(scHandlerID) != ERROR_ok) {
    printf("Error flushing client updates");
}
```

Example for doing two changes:

```
/* Modify data 1 */
if(ts3client_setClientSelfVariableAsInt(scHandlerID, CLIENT_AWAY, AWAY_ZZZ) != ERROR_ok) {
    printf("Error setting away mode\n");
    return;
}

/* Modify data 2 */
if(ts3client_setClientSelfVariableAsString(scHandlerID, CLIENT_AWAY_MESSAGE, "Lunch") != ERROR_ok) {
    printf("Error setting away message\n");
    return;
}

/* Flush changes */
if(ts3client_flushClientSelfUpdates(scHandlerID) != ERROR_ok) {
    printf("Error flushing client updates");
}
```

Example to mute and unmute the microphone:

```
unsigned int error;
bool shouldTalk;

shouldTalk = isPushToTalkButtonPressed();  // Your key detection implementation
if((error = ts3client_setClientSelfVariableAsInt(scHandlerID, CLIENT_INPUT_DEACTIVATED,
                                                 shouldTalk ? INPUT_ACTIVE : INPUT_DEACTIVATED)) != ERROR_ok) {
    char* errorMsg;
    if(ts3client_getErrorMessage(error, &errorMsg) != ERROR_ok) {
        printf("Error toggling push-to-talk: %s\n", errorMsg);
 ts3client_freeMemory(errorMsg);
    }
    return;
}

if(ts3client_flushClientSelfUpdates(scHandlerID) != ERROR_ok) {
    char* errorMsg;
    if(ts3client_getErrorMessage(error, &errorMsg) != ERROR_ok) {
        printf("Error flushing after toggling push-to-talk: %s\n", errorMsg);
 ts3client_freeMemory(errorMsg);
    }
}
```

See the FAQ section for further details on implementing Push-To-Talk with `ts3client_setClientSelfVariableAsInt`.

Information related to other clients

Information related to other clients can be retrieved in a similar way. Unlike own clients however, information cannot be modified.

To query client related information, use one of the following functions. The parameter *`flag`* is defined by the enum Client-Properties as shown above.

```
unsigned int ts3client_getClientVariableAsInt(serverConnectionHandlerID, clientID,
flag, result);

anyID serverConnectionHandlerID;
anyID clientID;
ClientProperties flag;
int* result;
```

```
unsigned int ts3client_getClientVariableAsString(serverConnectionHandlerID, clien-
tID, flag, result);

anyID serverConnectionHandlerID;
anyID clientID;
ClientProperties flag;
char** result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the information for the specified client is requested.

- *clientID*

  ID of the client whose property is queried.

- *flag*

  Client propery to query, see above.

- *result*

  Address of a variable which receives the result value as int or string, depending on which function is used. In case of a string, memory must be released using `ts3client_freeMemory`, unless an error occured.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. For the string version: If an error has occured, the result string is uninitialized and must not be released.

As the Client Lib cannot have all information for all users available all the time, the latest data for a given client can be requested from the server with:

```
unsigned int ts3client_requestClientVariables(serverConnectionHandlerID, clientID,
returnCode);

anyID serverConnectionHandlerID;
anyID clientID;
```

```
const char* returnCode;
```

The function requires one second delay before calling it again on the same client ID to avoid flooding the server.

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the client variables are requested.

- *clientID*

  ID of the client whose variables are requested.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After requesting the information, the following event is called:

```
void onUpdateClientEvent(serverConnectionHandlerID, clientID);

anyID serverConnectionHandlerID;
anyID clientID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the client variables are now available.

- *clientID*

  ID of the client whose variables are now available.

The event does not carry the information per se, but now the Client Lib guarantees to have the clients information available, which can be subsequently queried with ts3client_getClientVariableAsInt and ts3client_getClientVariableAsString.

Whisper lists

A client with a whisper list set can talk to the specified clients and channels bypassing the normal rule that voice is only transmitted to the current channel. Whisper lists can be defined for individual clients. A whisper list consists of an array of client IDs and/or an array of channel IDs.

```
unsigned int ts3client_requestClientSetWhisperList(serverConnectionHandlerID, clientID, targetChannelIDArray, targetClientIDArray, returnCode);

anyID serverConnectionHandlerID;
anyID clientID;
const anyID* targetChannelIDArray;
const anyID* targetClientIDArray;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the clients whisper list is modified.

- *clientID*

  ID of the client whose whisper list is modified. If set to 0, the own client is modified (same as setting to own client ID).

- *targetChannelIDArray*

  NULL-terminated array of channel IDs. These channels will be added to the whisper list.

  To clear the list, pass NULL or an empty array.

- *targetClientIDArray*

  NULL-termianted array of client IDs. These clients will be added to the whisper list.

  To clear the list, pass NULL or an empty array.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

To disable the whisperlist for the given client, pass NULL to both *targetChannelIDArray* and *targetClientIDArray*. Careful: If you pass two empty arrays, whispering is *not* disabled but instead one would still be whispering to nobody (empty lists).

Channel information

Querying and modifying information related to channels is similar to dealing with clients. The functions to query channel information are:

```
unsigned int ts3client_getChannelVariableAsInt(serverConnectionHandlerID, channelID, flag, result);

anyID serverConnectionHandlerID;
anyID channelID;
```

```
ChannelProperties flag;
int* result;
```

```
unsigned int ts3client_getChannelVariableAsString(serverConnectionHandlerID, chan-
nelID, flag, result);

anyID serverConnectionHandlerID;
anyID channelID;
ChannelProperties flag;
char* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the information for the specified channel is requested.

- *channelID*

  ID of the channel whose property is queried.

- *flag*

  Channel propery to query, see below.

- *result*

  Address of a variable which receives the result value as int or string, depending on which function is used. In case of a string, memory must be released using `ts3client_freeMemory`, unless an error occured.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. For the string version: If an error has occured, the result string is uninitialized and must not be released.

The parameter *flag* specifies the type of queried information. It is defined by the enum ChannelProperties:

```
enum ChannelProperties {
  CHANNEL_NAME = 0,             //Available for all channels that are "in view", always up-to-date
  CHANNEL_TOPIC,                //Available for all channels that are "in view", always up-to-date
  CHANNEL_DESCRIPTION,          //Must be requested (=> requestChannelDescription)
  CHANNEL_PASSWORD,             //not available client side
  CHANNEL_CODEC,                //Available for all channels that are "in view", always up-to-date
  CHANNEL_CODEC_QUALITY,        //Available for all channels that are "in view", always up-to-date
  CHANNEL_MAXCLIENTS,           //Available for all channels that are "in view", always up-to-date
  CHANNEL_MAXFAMILYCLIENTS,     //Available for all channels that are "in view", always up-to-date
  CHANNEL_ORDER,                //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_PERMANENT,       //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_SEMI_PERMANENT,  //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_DEFAULT,         //Available for all channels that are "in view", always up-to-date
  CHANNEL_FLAG_PASSWORD,        //Available for all channels that are "in view", always up-to-date
  CHANNEL_ENDMARKER,
};
```

- *CHANNEL_NAME*

String: Name of the channel.

- *CHANNEL_TOPIC*

  String: Single-line channel topic.

- *CHANNEL_DESCRIPTION*

  String: Optional channel description. Can have multiple lines. Clients need to request updating this variable for a specified channel using:

```
unsigned int ts3client_requestChannelDescription(serverConnectionHandlerID, channelID, returnCode);

anyID serverConnectionHandlerID;
anyID channelID;
const char* returnCode;
```

- *CHANNEL_PASSWORD*

  String: Optional password for password-protected channels.

Note

Clients can only *set* this value, but not query it.

  If a password is set or removed by modifying this field, *CHANNEL_FLAG_PASSWORD* will be automatically adjusted.

- *CHANNEL_CODEC*

  Int (0-3): Codec used for this channel:

  - 0 - Speex Narrowband (8 kHz)
  - 1 - Speex Wideband (16 kHz)
  - 2 - Speex Ultra-Wideband (32 kHz)

  See Sound codecs.

- *CHANNEL_CODEC_QUALITY*

  Int (0-10): Quality of channel codec of this channel. Valid values range from 0 to 10, default is 7. Higher values result in better speech quality but more bandwidth usage.

  See Encoder options.

- *CHANNEL_MAXCLIENTS*

  Int: Number of maximum clients who can join this channel.

- *CHANNEL_MAXFAMILYCLIENTS*

Int: Number of maximum clients who can join this channel and all subchannels.

- *CHANNEL_ORDER*

  Int: Defines how channels are sorted in the GUI. Channel order is the ID of the predecessor channel after which this channel is to be sorted. If 0, the channel is sorted at the top of its hirarchy.

  For more information please see the chapter Channel sorting.

- *CHANNEL_FLAG_PERMANENT* / *CHANNEL_FLAG_SEMI_PERMANENT*

  Concerning channel durability, there are three types of channels:

  - Temporary

    Temporary channels have neither the *CHANNEL_FLAG_PERMANENT* nor *CHANNEL_FLAG_SEMI_PERMANENT* flag set. Temporary channels are automatically deleted by the server after the last user has left and the channel is empty. They will not be restored when the server restarts.

  - Semi-permanent

    Semi-permanent channels are not automatically deleted when the last user left but will not be restored when the server restarts.

  - Permanent

    Permanent channels will be restored when the server restarts.

- *CHANNEL_FLAG_DEFAULT*

  Int (0/1): Channel is the default channel. There can only be one default channel per server. New users who did not configure a channel to join on login in `ts3client_startConnection` will automatically join the default channel.

- *CHANNEL_FLAG_PASSWORD*

  Int (0/1): If set, channel is password protected. The password itself is stored in *CHANNEL_PASSWORD*.

To modify channel data use

```
unsigned int ts3client_setChannelVariableAsInt(serverConnectionHandlerID, channelID,
flag, value);

anyID serverConnectionHandlerID;
anyID channelID;
ChannelProperties flag;
int value;
```

```
unsigned int ts3client_setChannelVariableAsString(serverConnectionHandlerID, chan-
nelID, flag, value);
```

```
anyID serverConnectionHandlerID;
anyID channelID;
ChannelProperties flag;
const char* value;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the information for the specified channel should be changed.

- *channelID*

  ID of the channel whoses property should be changed.

- *flag*

  Channel propery to change, see above.

- *value*

  Value the channel property should be changed to.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Important

After modifying one or more channel variables, you have to flush the changes to the server.

```
unsigned int ts3client_flushChannelUpdates(serverConnectionHandlerID, channelID);

anyID serverConnectionHandlerID;
anyID channelID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler to which the channel changes should be flushed.

- *channelParentID*

  ID of the channel of which the changed properties should be flushed.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

As example, to change the channel name and topic:

```
/* Modify data 1 */
if(ts3client_setChannelVariableAsString(scHandlerID, channelID, CHANNEL_NAME,
```

```
                                        "Other channel name") != ERROR_ok) {
    printf("Error setting channel name\n");
    return;
}

/* Modify data 2 */
if(ts3client_setChannelVariableAsString(scHandlerID, channelID, CHANNEL_TOPIC,
                                        "Other channel topic") != ERROR_ok) {
printf("Error setting channel topic\n");
    return;
}

/* Flush changes */
if(ts3client_flushChannelUpdates(scHandlerID, channelID) != ERROR_ok) {
    printf("Error flushing channel updates\n");
    return;
}
```

After a channel was edited using `ts3client_setChannelVariableAsInt` or `ts3client_setChannelVariableAsString` and the changes were flushed to the server, the edit is announced with the event:

`void` **`onUpdateChannelEditedEvent`**`(`*`serverConnectionHandlerID`*`, `*`channelID`*`, `*`invokerID`*`, `*`invokerName`*`, `*`invokerUniqueIdentifier`*`);`

```
anyID serverConnectionHandlerID;
anyID channelID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

Parameters

- *`serverConnectionHandlerID`*

  ID of the server connection handler on which the channel has been edited.

- *`channelID`*

  ID of edited channel.

- *`invokerID`*

  ID of the client who edited the channel.

- *`invokerName`*

  String with the name of the client who edited the channel.

- *`invokerUniqueIdentifier`*

  String with the unique ID of the client who edited the channel.

To find the channel ID from a channels path:

```
unsigned int ts3client_getChannelIDFromChannelNames(serverConnectionHandlerID, channelNameArray, result);

anyID serverConnectionHandlerID;
char** channelNameArray;
anyID* result;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the channel ID is queried.

- *channelNameArray*

  Array defining the position of the channel: "grandparent", "parent", "channel", "". The array is terminated by an empty string.

- *result*

  Address of a variable which receives the queried channel ID.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

Channel sorting

The order how channels should be display in the GUI is defined by the channel variable *CHANNEL_ORDER*, which can be queried with `ts3client_getChannelVariableAsInt` or changed with `ts3client_setChannelVariableAsInt`.

The channel order is the ID of the predecessor channel after which the given channel should be sorted. An order of 0 means the channel is sorted on the top of its hirarchy.

Channel_1 (ID = 1, order = 0)
Channel_2 (ID = 2, order = 1)
    Subchannel_1 (ID = 4, order = 0)
        Subsubchannel_1 (ID = 6, order = 0)
        Subsubchannel_2 (ID = 7, order = 6)
    Subchannel_2 (ID = 5, order = 4)
Channel_3 (ID = 3, order = 2)

When a new channel is created, the client is responsible to set a proper channel order. With the default value of 0 the channel will be sorted on the top of its hirarchy right after its parent channel.

When moving a channel to a new parent, the desired channel order can be passed to `ts3client_requestChannelMove`.

To move the channel to another position within the current hirarchy - the parent channel stays the same -, adjust the *CHANNEL_ORDER* variable with `ts3client_setChannelVariableAsInt`.

After connecting to a TeamSpeak 3 server, the client will be informed of all channels by the `onNewChannelEvent` callback. The order how channels are propagated to the client by this event is:

- First the complete channel path to the default channel, which is either the servers default channel with the flag *CHANNEL_FLAG_DEFAULT* or the users default channel passed to `ts3client_startConnection`. This ensures the channel joined on login is visible as soon as possible.

  In above example, assuming the default channel is “Subsubchannel_2”, the channels would be announced in the following order: Channel_2, Subchannel_1, Subsubchannel_2.

  After the default channel path has completely arrived, the connection status (see enum ConnectStatus, annouced to the client by the callback `onConnectStatusChangeEvent`) changes to *STATUS_CONNECTION_ESTABLISHING*.

- Next all other channels in the given order, where subchannels are announced right after the parent channel.

  To continue the example, the remaining channels would be announced in the order of: Channel_1, Subsubchannel_1, Subchannel_2, Channel_3 (Channel_2, Subchannel_1, Subsubchannel_2 already were announced in the previous step).

  When all channels have arrived, the connection status switches to *STATUS_CONNECTION_ESTABLISHED*.

Server information

Similar to querying client and channel data, server information can be checked with

```
unsigned int ts3client_getServerVariableAsInt(serverConnectionHandlerID, flag, result);

anyID serverConnectionHandlerID;
VirtualServerProperties flag;
int* result;
```

```
unsigned int ts3client_getServerVariableAsUInt64(serverConnectionHandlerID, flag, result);

anyID serverConnectionHandlerID;
VirtualServerProperties flag;
uint64* result;
```

```
unsigned int ts3client_getServerVariableAsString(serverConnectionHandlerID, flag, result);

anyID serverConnectionHandlerID;
VirtualServerProperties flag;
char** result;
```

Parameters

- *serverConnectionHandlerID*

ID of the server connection handler on which the virtual server property is queried.

- *clientID*

  ID of the client whose property is queried.

- *flag*

  Virtual server propery to query, see below.

- *result*

  Address of a variable which receives the result value as int, uint64 or string, depending on which function is used. In case of a string, memory must be released using `ts3client_freeMemory`, unless an error occured.

  The returned type uint64 is defined as __int64 on Windows and uint64_t on Linux and Mac OS X. See the header `public_definitions.h`. This function is currently only used for the flag *VIRTUALSERVER_UPTIME*.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`. For the string version: If an error has occured, the result string is uninitialized and must not be released.

The parameter *flag* specifies the type of queried information. It is defined by the enum VirtualServerProperties:

```
enum VirtualServerProperties {
  VIRTUALSERVER_UNIQUE_IDENTIFIER = 0, //available when connected, can be used to identify this particular
                                       //server installation
  VIRTUALSERVER_NAME,                  //available and always up-to-date when connected
  VIRTUALSERVER_WELCOMEMESSAGE,        //available when connected, not updated while connected
  VIRTUALSERVER_PLATFORM,              //available when connected
  VIRTUALSERVER_VERSION,               //available when connected
  VIRTUALSERVER_MAXCLIENTS,            //only available on request (=> requestServerVariables), stores the
                                       //maximum number of clients that may currently join the server
  VIRTUALSERVER_PASSWORD,              //not available to clients, the server password
  VIRTUALSERVER_CLIENTS_ONLINE,        //only available on request (=> requestServerVariables),
  VIRTUALSERVER_CHANNELS_ONLINE,       //only available on request (=> requestServerVariables),
  VIRTUALSERVER_CREATED,               //available when connected, stores the time when the server was created
  VIRTUALSERVER_UPTIME,                //only available on request (=> requestServerVariables), the time
                                       //since the server was started
  VIRTUALSERVER_ENDMARKER,
};
```

- *VIRTUALSERVER_UNIQUE_IDENTIFIER*

  Unique ID for this virtual server. Stays the same after restarting the server application. Always available when connected.

- *VIRTUALSERVER_NAME*

  Name of this virtual server. Always available when connected.

- *VIRTUALSERVER_WELCOMEMESSAGE*

  Optional welcome message sent to the client on login. This value should be queried by the client after connection has been established, it is *not* updated afterwards.

- *VIRTUALSERVER_PLATFORM*

  Operating system used by this server. Always available when connected.

- *VIRTUALSERVER_VERSION*

Application version of this server. Always available when connected.

- *VIRTUALSERVER_MAXCLIENTS*

  Defines maximum number of clients which may connect to this server. Needs to be requested using `ts3client_requestServerVariables`.

- *VIRTUALSERVER_PASSWORD*

  Optional password of this server. Not available to clients.

- *VIRTUALSERVER_CLIENTS_ONLINE*

  VIRTUALSERVER_CHANNELS_ONLINE

  Number of clients and channels currently on this virtual server. Needs to be requested using `ts3client_requestServerVariables`.

- *VIRTUALSERVER_CREATED*

  Time when this virtual server was created. Always available when connected.

- *VIRTUALSERVER_UPTIME*

  Uptime of this virtual server. Needs to be requested using `ts3client_requestServerVariables`.

Example code checking the number of clients online, obviously an integer value:

```
int clientsOnline;

if(ts3client_getServerVariableAsInt(scHandlerID, VIRTUALSERVER_CLIENTS_ONLINE, &clientsOnline) == ERROR_ok)
    printf("There are %d clients online\n", clientsOnline);
```

A client can request refreshing the server information with:

```
unsigned int ts3client_requestServerVariables(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

The following event informs the client when the requested information is available:

```
unsigned int onServerUpdatedEvent(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

The following event notifies the client when virtual server information has been edited:

```
void onServerEditedEvent(serverConnectionHandlerID, editerID, editerName,
editerUniqueIdentifier);

anyID serverConnectionHandlerID;
anyID editerID;
anyID editerName;
const char* editerUniqueIdentifier;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler which virtual server information has been changed.

- *editerID*

  ID of the client who edited the information. If zero, the server is the editor.

- *editerName*

  Name of the client who edited the information.

- *editerUniqueIdentifier*

  Unique ID of the client who edited the information.

Interacting with the server

Interacting with the server means various actions, related to both channels and clients. Channels can be joined, created, edited, deleted and subscribed. Clients can use text chat with other clients, be kicked or poked and move between channels.

All strings passed to and from the Client Lib need to be encoded in UTF-8 format.

Joining a channel

When a client logs on to a TeamSpeak 3 server, he will automatically join the channel with the “Default” flag, unless he specified another channel in `ts3client_startConnection`. To have your own or another client switch to a certain channel, call

```
unsigned int ts3client_requestClientMove(serverConnectionHandlerID, clID, newChan-
nelID, password, returnCode);

anyID serverConnectionHandlerID;
anyID clID;
anyID newChannelID;
const char* password;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler ID on which this action is requested.

- *clID*

  ID of the client to move.

- *newChannelID*

  ID of the channel the client wants to join.

- *password*

  An optional password, required for password-protected channels.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

If the move was successful, one the following events will be called:

```
void onClientMoveEvent(serverConnectionHandlerID, clientID, oldChannelID, newChan-
nelID, visibility, moveMessage);

anyID serverConnectionHandlerID;
anyID clientID;
anyID oldChannelID;
anyID newChannelID;
int visibility;
const char* moveMessage;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the action occured.

- *clientID*

  ID of the moved client.

- *oldChannelID*

ID of the old channel left by the client.

- *newChannelID*

  ID of the new channel joined by the client.

- *visibility*

  Defined in the enum Visibility

```
enum Visibility {
    ENTER_VISIBILITY = 0,
    RETAIN_VISIBILITY,
    LEAVE_VISIBILITY
};
```

  - *ENTER_VISIBILITY*

    Client moved and entered visibility. Cannot happen on own client.

  - *RETAIN_VISIBILITY*

    Client moved between two known places. Can happen on own or other client.

  - *LEAVE_VISIBILITY*

    Client moved out of our sight. Cannot happen on own client.

- *moveMessage*

  Displaying the optional message given in `ts3client_stopConnection`.

Example: Requesting to move the own client into channel ID 12 (not password-protected):

```
ts3client_requestClientMove(scHandlerID, ts3client_getClientID(scHandlerID), 12, "");
```

Now wait for the callback:

```
void yourImplementationOf_onClientMoveEvent(anyID scHandlerID, anyID clientID,
                                            anyID oldChannelID, anyID newChannelID,
                                            int visibility) {
  // scHandlerID   ->  Server connection handler ID, same as above when requesting
  // clientID      ->  Own client ID, same as above when requesting
  // oldChannelID  ->  ID of the channel the client has left
  // newChannelID  ->  12, as requested above
  // visibility    ->  One of ENTER_VISIBILITY, RETAIN_VISIBILITY, LEAVE_VISIBILITY
}
```

If the move was initiated by another client, instead of `onClientMove` the following event is called:

```
void onClientMoveMovedEvent(serverConnectionHandlerID, clientID, oldChannelID,
newChannelID, visibility, moverID, moverName, moverUniqueIdentifier, moveMessage);

anyID serverConnectionHandlerID;
anyID clientID;
anyID oldChannelID;
```

```
anyID newChannelID;
int visibility;
anyID moverID;
const char* moverName;
moveMessage moverUniqueIdentifier;
moveMessage moveMessage;
```

Like `onClientMoveEvent` but with additional information about the client, which has initiated the move: *`moverID`* defines the ID, *`moverName`* the nickname and *`moverUniqueIdentifier`* the unique ID of the mover client. *`moveMessage`* contains a string giving the reason for the move.

If *`oldChannelID`* is 0, the client has just connected to the server. If *`newChannelID`* is 0, the client disconnected. Both values cannot be 0 at the same time.

Creating a new channel

To create a channel, set the various channel variables using `ts3client_setChannelVariableAsInt` and `ts3client_setChannelVariableAsString`. Pass zero as the channel ID parameter.

Then flush the changes to the server by calling:

```
unsigned int ts3client_flushChannelCreation(serverConnectionHandlerID, channelParentID);

anyID serverConnectionHandlerID;
anyID channelParentID;
```

Parameters

- *`serverConnectionHandlerID`*

  ID of the server connection handler to which the channel changes should be flushed.

- *`channelParentID`*

  ID of the parent channel, if the new channel is to be created as subchannel. Pass zero if the channel should be created as top-level channel.

Returns *`ERROR_ok`* on success, otherwise an error code as defined in `public_errors.h`.

After flushing the changes to the server, the following event will be called on successful channel creation:

```
void onNewChannelCreatedEvent(serverConnectionHandlerID, channelID, channelParentID, invokerID, invokerName, invokerUniqueIdentifier);

anyID serverConnectionHandlerID;
anyID channelID;
```

```
anyID channelParentID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler where the channel was created.

- *channelID*

  ID of the created channel. Channel IDs start with the value 1.

- *channelParentID*

  ID of the parent channel.

- *invokerID*

  ID of the client who requested the creation. If zero, the request was initiated by the server.

- *invokerName*

  Name of the client who requested the creation. If requested by the server, the name is empty.

- *invokerUniqueIdentifier*

  Unique ID of the client who requested the creation.

Example code to create a channel:

```
#define CHECK_ERROR(x) if((error = x) != ERROR_ok) { goto on_error; }

int createChannel(anyID scHandlerID, anyID parentChannelID, const char* name, const char* topic,
                  const char* description, const char* password, int codec, int codecQuality,
                  int maxClients, int familyMaxClients, int order, int perm,
                  int semiperm, int default) {
  unsigned int error;

  /* Set channel data, pass 0 as channel ID */
  CHECK_ERROR(ts3client_setChannelVariableAsString(scHandlerID, 0, CHANNEL_NAME, name));
  CHECK_ERROR(ts3client_setChannelVariableAsString(scHandlerID, 0, CHANNEL_TOPIC, topic));
  CHECK_ERROR(ts3client_setChannelVariableAsString(scHandlerID, 0, CHANNEL_DESCRIPTION, desc));
  CHECK_ERROR(ts3client_setChannelVariableAsString(scHandlerID, 0, CHANNEL_PASSWORD, password));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_CODEC, codec));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_CODEC_QUALITY, codecQuality));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_MAXCLIENTS, maxClients));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_MAXFAMILYCLIENTS, familyMaxClients));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_ORDER, order));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_FLAG_PERMANENT, perm));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_FLAG_SEMI_PERMANENT, semiperm));
  CHECK_ERROR(ts3client_setChannelVariableAsInt   (scHandlerID, 0, CHANNEL_FLAG_DEFAULT, default));
```

```
  /* Flush changes to server */
  CHECK_ERROR(ts3client_flushChannelCreation(scHandlerID, parentChannelID));
  return 0;  /* Success */

on_error:
  printf("Error creating channel: %d\n", error);
  return 1;  /* Failure */
}
```

Deleting a channel

A channel can be removed with

```
unsigned int ts3client_requestChannelDelete(serverConnectionHandlerID, channelID, force, returnCode);

anyID serverConnectionHandlerID;
anyID channelID;
int force;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the channel should be deleted.

- *channelID*

  The ID of the channel to be deleted.

- *force*

  If 1, the channel will be deleted even when it is not empty. Clients within the deleted channel are transfered to the default channel. Any contained subchannels are removed as well.

  If 0, the server will refuse to a channel that is not empty.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in `public_errors.h`.

After the request has been sent to the server, the following event will be called:

```
void onDelChannelEvent(serverConnectionHandlerID, channelID, invokerID, invokerName, invokerUniqueIdentifier);

anyID serverConnectionHandlerID;
```

```
anyID channelID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the channel was deleted.

- *channelID*

  The ID of the deleted channel.

- *invokerID*

  The ID of the client who requested the deletion. If zero, the deletion was initiated by the server (for example automatic deletion of empty non-permanent channels).

- *invokerName*

  The name of the client who requested the deletion. Empty if requested by the server.

- *invokerUniqueIdentifier*

  The unique ID of the client who requested the deletion.

Moving a channel

To move a channel to a new parent channel, call

```
unsigned int ts3client_requestChannelMove(serverConnectionHandlerID, channelID, newChannelParentID, newChannelOrder, returnCode);

anyID serverConnectionHandlerID;
anyID channelID;
anyID newChannelParentID;
anyID newChannelOrder;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the channel should be moved.

- *channelID*

ID of the channel to be moved.

- *newChannelParentID*

  ID of the parent channel where the moved channel is to be inserted as child. Use *0* to insert as top-level channel.

- *newChannelOrder*

  Channel order defining where the channel should be sorted under the new parent. Pass 0 to sort the channel right after the parent. See the chapter Channel sorting for details.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After sending the request, the following event will be called if the move was successful:

```
void onChannelMoveEvent(serverConnectionHandlerID, channelID, newChannelParentID,
invokerID, invokerName, invokerUniqueIdentifier);

anyID serverConnectionHandlerID;
anyID channelID;
anyID newChannelParentID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the channel was moved.

- *channelID*

  The ID of the moved channel.

- *newChannelParentID*

  ID of the parent channel where the moved channel is inserted as child. *0* if inserted as top-level channel.

- *invokerID*

  The ID of the client who requested the move. If zero, the move was initiated by the server.

- *invokerName*

The name of the client who requested the move. Empty if requested by the server.

- *invokerUniqueIdentifier*

  The unique ID of the client who requested the move.

Text chat

In addition to voice chat, TeamSpeak 3 allows clients to communicate with text-chat. Valid targets can be single and multiple clients and channels or the whole server.

Sending

To send a text message, call

```
unsigned int ts3client_requestSendTextMsg(serverConnectionHandlerID, targetMode, message, targetID, returnCode);

anyID serverConnectionHandlerID;
int targetMode;
const char* message;
const anyID* targetID;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

  Id of the target server connection handler.

- *targetMode*

  Sets the target type of the sent message. The value is defined by the enum TextMessageTargetMode:

  ```
  enum TextMessageTargetMode {
      TextMessageTarget_CLIENT=1,
      TextMessageTarget_CHANNEL,
      TextMessageTarget_SERVER,
      TextMessageTarget_MAX
  };
  ```

- *message*

  String containing the text message

- *targetID*

  NULL-terminated array of target IDs. The type of the given IDs depends on the *targetMode*. If the target mode is for example *TextMessageTarget_CLIENT*, the target IDs will be treated as client IDs.

- *returnCode*

See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Example to send a text chat to a single client with the ID 123:

```
const char *msg = "Hello TeamSpeak!";
anyID targetIDList[2];
targetIDList[0] = 123;  /* Client ID in this example */
targetIDList[1] = NULL; /* NULL-terminated */

if(ts3client_requestSendTextMsg(scHandlerID, TextMessageTarget_Client, msg, targetIDList) != ERROR_ok) {
  /* Handle error */
}
```

Receiving

The following event will be called when a text message is received:

```
void onTextMessageEvent(serverConnectionHandlerID, targetMode, fromID, fromName,
fromUniqueIdentifier, message, targets);

anyID serverConnectionHandlerID;
anyID targetMode;
anyID fromID;
const char* fromName;
const char* fromUniqueIdentifier;
const char* message;
anyID* targets;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler from which the text message was sent.

- *targetMode*

  The requested targetMode, defined by the enum TextMessageTargetMode.

- *fromID*

  Id of the client who sent the text message.

- *fromName*

  Name of the client who sent the text message.

- *fromUniqueIdentifier*

Unique ID of the client who sent the text message.

- *message*

  String containing the text message.

- *targets*

  NULL-terminated array of target IDs. The type of the given IDs depends on the targetMode.

Kicking clients

Clients can be forcefully removed from a channel or the whole server. To kick a client from a channel or server call:

```
unsigned int ts3client_requestClientKickFromChannel(serverConnectionHandlerID, clID,
kickReason, returnCode);

anyID serverConnectionHandlerID;
anyID clID;
const char* kickReason;
const char* returnCode;
```

```
unsigned int ts3client_requestClientKickFromServer(serverConnectionHandlerID, clID,
kickReason, returnCode);

anyID serverConnectionHandlerID;
anyID clID;
const char* kickReason;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

  Id of the target server connection.

- *clID*

  The ID of the client to be kicked.

- *kickReason*

  A short message explaining why the client is kicked from the channel or server.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

After successfully requesting a kick, one of the following events will be called:

```
void onClientKickFromChannelEvent(serverConnectionHandlerID, clientID, oldChannelID,
newCHannelID, visibility, kickerID, kickerName, kickerUniqueIdentifier, kickMes-
sage);

anyID serverConnectionHandlerID;
anyID clientID;
anyID oldChannelID;
anyID newCHannelID;
int visibility;
anyID kickerID;
const char* kickerName;
const char* kickerUniqueIdentifier;
const char* kickMessage;
```

```
void onClientKickFromServerEvent(serverConnectionHandlerID, clientID, oldChannelID,
newChannelID, visibility, kickerID, kickerName, kickerUniqueIdentifier, kickMes-
sage);

anyID serverConnectionHandlerID;
anyID clientID;
anyID oldChannelID;
anyID newChannelID;
int visibility;
anyID kickerID;
const char* kickerName;
const char* kickerUniqueIdentifier;
const char* kickMessage;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the client was kicked

- *clientID*

  ID of the kicked client.

- *oldChannelID*

  ID of the channel from which the client has been kicked.

- *newChannelID*

ID of the channel where the kicked client was moved to.

- *visibility*

  Describes if the moved client enters, retains or leaves visibility. See explanation of the enum Visibility for the function `onClientMoveEvent`.

  When kicked from a server, visibility can be only *LEAVE_VISIBILITY*.

- *kickerID*

  ID of the client who requested the kick.

- *kickerName*

  Name of the client who requested the kick.

- *kickerUniqueIdentifier*

  Unique ID of the client who requested the kick.

- *kickerMessage*

  Message giving the reason why the client has been kicked.

Channel subscriptions

Normally a user only sees other clients who are in the same channel. Clients joining or leaving other channels or changing status are not displayed. To offer a way to get notifications about clients in other channels, a user can subscribe to other channels.

Note

A client is automatically subscribed to a joined channel.

To subscribe to a channel call:

```
unsigned int ts3client_requestChannelSubscribe(serverConnectionHandlerID, channelID,
returnCode);

anyID serverConnectionHandlerID;
anyID channelID;
const char* returnCode;
```

To unsubscribe to a single channel call:

```
unsigned int ts3client_requestChannelUnsubscribe(serverConnectionHandlerID, chan-
nelID, returnCode);

anyID serverConnectionHandlerID;
anyID channelID;
```

```
const char* returnCode;
```

To subscribe to all channels on the server call:

```
unsigned int ts3client_requestChannelSubscribeAll(serverConnectionHandlerID, return-
Code);

anyID serverConnectionHandlerID;
const char* returnCode;
```

To unsubscribe to all channels on the server call:

```
unsigned int ts3client_requestChannelUnsubscribeAll(serverConnectionHandlerID, re-
turnCode);

anyID serverConnectionHandlerID;
const char* returnCode;
```

To check if a channel is currently subscribed, check the flag *CHANNEL_FLAG_ARE_SUBSCRIBED* with `ts3client_getChannelVariableAsInt`:

```
int isSubscribed;

if(ts3client_getChannelVariableAsInt(scHandlerID, channelID, CHANNEL_FLAG_ARE_SUBSCRIBED, &isSubscribed)
   != ERROR_ok) {
    /* Handle error */
}
```

The following event will be sent for each successfully subscribed channel:

```
void onChannelSubscribeEvent(serverConnectionHandlerID, channelID);

anyID serverConnectionHandlerID;
anyID channelID;
```

Provided for convinience, to mark the end of mulitple calls to `onChannelSubscribeEvent` when subscribing to several channels, this event is called:

```
void onChannelSubscribeFinishedEvent(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

The following event will be sent for each successfully unsubscribed channel:

```
void onChannelUnsubscribeEvent(serverConnectionHandlerID, channelID);

anyID serverConnectionHandlerID;
anyID channelID;
```

Similar like subscribing, this event is a convinience callback to mark the end of multiple calls to `onChannelUnsubscribeEvent`:

```
void onChannelUnsubscribeFinishedEvent(serverConnectionHandlerID);

anyID serverConnectionHandlerID;
```

Once a channel has been subscribed or unsubscribed, the event `onClientMoveSubscriptionEvent` is sent for each client in the subscribed channel. The event is not to be confused with `onClientMoveEvent`, which is called for clients actively switching channels.

```
void onClientMoveSubscriptionEvent(serverConnectionHandlerID, clientID, oldChannelID, newChannelID, visibility);

anyID serverConnectionHandlerID;
anyID clientID;
anyID oldChannelID;
anyID newChannelID;
int visibility;
```

Parameters

- *serverConnectionHandlerID*

  The server connection handler ID for the server where the action occured.

- *clientID*

  The client ID.

- *oldChannelID*

  ID of the subscribed channel where the client left visibility.

- *newChannelID*

  ID of the subscribed channel where the client entered visibility.

- *visibility*

  Defined in the enum Visibility

  ```
  enum Visibility {
  ```

```
    ENTER_VISIBILITY = 0,
    RETAIN_VISIBILITY,
    LEAVE_VISIBILITY
};
```

- *ENTER_VISIBILITY*

  Client entered visibility.

- *LEAVE_VISIBILITY*

  Client left visibility.

- *RETAIN_VISIBILITY*

  Does not occur with onClientMoveSubscriptionEvent.

Muting clients locally

Individual clients can be locally muted. This information is handled client-side only and not visibile to other clients. It mainly serves as a sort of individual "ban" or "ignore" feature, where users can decide not to listen to certain clients anymore.

When a client becomes muted, he will no longer be heard by the muter. Also the TeamSpeak 3 server will stop sending voice packets.

The mute state is not visible to the muted client nor to other clients. It is only available to the muting client by checking the *CLIENT_IS_MUTED* client property.

To mute one or more clients:

```
unsigned int ts3client_requestMuteClients(serverConnectionHandlerID, clientIDArray, returnCode);

anyID serverConnectionHandlerID;
const anyID* clientIDArray;
const char* returnCode;
```

To unmute one or more clients:

```
unsigned int ts3client_requestUnmuteClients(serverConnectionHandlerID, clientIDArray, returnCode);

anyID serverConnectionHandlerID;
const anyID* clientIDArray;
const char* returnCode;
```

Parameters

- *serverConnectionHandlerID*

ID of the server connection handle on which the client should be locally (un)muted

- *clientIDArray*

  NULL-terminated array of client IDs.

- *returnCode*

  See return code documentation. Pass NULL if you do not need this feature.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

Example to mute two clients:

```
anyID clientIDArray[3];  // List of two clients plus terminating zero
clientIDArray[0] = 123;  // First client ID to mute
clientIDArray[1] = 456;  // Second client ID to mute
clientIDArray[2] = 0;    // Terminating zero

if(ts3client_requestMuteClients(scHandlerID, clientIDArray) != ERROR_ok)  /* Mute clients */
    printf("Error muting clients: %d\n", error);
```

To check if a client is currently muted, query the *CLIENT_IS_MUTED* client property:

```
int clientIsMuted;
if(ts3client_getClientVariableAsInt(scHandlerID, clientID, CLIENT_IS_MUTED, &clientIsMuted) != ERROR_ok)
    printf("Error querying client muted state\n);
```

Custom encryption

As an optional feature, the TeamSpeak 3 SDK allows users to implement custom encryption and decryption for all network traffic. Custom encryption replaces the default AES encryption implemented by the TeamSpeak 3 SDK. A possible reason to apply own encryption might be to make ones TeamSpeak 3 client/server incompatible to other SDK implementations.

Custom encryption must be implemented the same way in both the client and server.

Note

If you do not want to use this feature, just don't implement the two encryption callbacks.

To encrypt outgoing data, implement the callback:

```
void onCustomPacketEncryptEvent(dataToSend, sizeOfData);

char** dataToSend;
unsigned int* sizeOfData;
```

Parameters

- *dataToSend*

  Pointer to an array with the outgoing data to be encrypted.

Apply your custom encryption to the data array. If the encrypted data is smaller than sizeOfData, write your encrypted data into the existing memory of dataToSend. If your encrypted data is larger, you need to allocate memory and redirect the pointer dataToSend. You need to take care of freeing your own allocated memory yourself. The memory allocated by the SDK, to which dataToSend is originally pointing to, must not be freed.

- *sizeOfData*

  Pointer to an integer value containing the size of the data array.

To decrypt incoming data, implement the callback:

```
void onCustomPacketDecryptEvent(dataReceived, dataReceivedSize);

char** dataReceived;
unsigned int* dataReceivedSize;
```

Parameters

- *dataReceived*

  Pointer to an array with the received data to be decrypted.

  Apply your custom decryption to the data array. If the decrypted data is smaller than dataReceivedSize, write your decrypted data into the existing memory of dataReceived. If your decrypted data is larger, you need to allocate memory and redirect the pointer dataReceived. You need to take care of freeing your own allocated memory yourself. The memory allocated by the SDK, to which dataReceived is originally pointing to, must not be freed.

- *dataReceivedSize*

  Pointer to an integer value containing the size of the data array.

Example code implementing a very simple XOR custom encryption and decryption (also see the SDK examples):

```
void onCustomPacketEncryptEvent(char** dataToSend, unsigned int* sizeOfData) {
    unsigned int i;
    for(i = 0; i < *sizeOfData; i++) {
        (*dataToSend)[i] ^= CUSTOM_CRYPT_KEY;
    }
}

void onCustomPacketDecryptEvent(char** dataReceived, unsigned int* dataReceivedSize) {
    unsigned int i;
    for(i = 0; i < *dataReceivedSize; i++) {
        (*dataReceived)[i] ^= CUSTOM_CRYPT_KEY;
    }
}
```

Other events

When a client starts or stops talking, a talk status change event is sent by the server:

```
void onTalkStatusChangeEvent(serverConnectionHandlerID, status, clientID);

anyID serverConnectionHandlerID;
int status;
anyID clientID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the event occured.

- *status*

  Possible return values are defined by the enum TalkStatus:

  ```
  enum TalkStatus {
      STATUS_NOT_TALKING = 0,
      STATUS_TALKING = 1,
      STATUS_TALKING_WHILE_DISABLED = 2,
  };
  ```

  STATUS_TALKING and *STATUS_TALKING* are triggered everytime a client starts or stops talking. *STATUS_TALKING_WHILE_DISABLED* is triggered only if the microphone is muted. A client application might use this to implement a mechanism warning the user he is talking while not sending to the server.

- *clientID*

  ID of the client who started or stopped talking.

If a client drops his connection, a timeout event is announced by the server:

```
void onClientMoveTimeoutEvent(serverConnectionHandlerID, clientID, oldChannelID,
newChannelID, visibility, timeoutMessage);

anyID serverConnectionHandlerID;
anyID clientID;
anyID oldChannelID;
anyID newChannelID;
int visibility;
const char* timeoutMessage;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the event occured.

- *clientID*

  ID of the moved client.

- *oldChannelID*

  ID of the channel the leaving client was previously member of.

- *newChannelID*

  0, as client is leaving.

- *visibility*

  Always *LEAVE_VISIBILITY*.

- *timeoutMessage*

  Optional message giving the reason for the timeout. UTF-8 encoded.

When the description of a channel was edited, the following event is called:

```
void onChannelDescriptionUpdateEvent(serverConnectionHandlerID, channelID);

anyID serverConnectionHandlerID;
anyID channelID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the event occured.

- *shutdownMessage*

  ID of the channel with the edited description.

The new description can be queried with `ts3client_getChannelVariableAsString(channelID, CHANNEL_DESCRIPTION)`.

This event tells the client that the specified channel has been modified. The GUI should fetch the channel data with `ts3client_getChannelVariableAsInt` and `ts3client_getChannelVariableAsString` and update the channel display.

```
void onUpdateChannelEvent(serverConnectionHandlerID, channelID);

anyID serverConnectionHandlerID;
```

```
anyID channelID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the event occured.

- *channelID*

  ID of the updated channel.

The following event is called when a channel password was modified. The GUI might remember previously entered channel passwords, so this callback announces the stored password might be invalid.

```
void onChannelPasswordChangedEvent(serverConnectionHandlerID, channelID);

anyID serverConnectionHandlerID;
anyID channelID;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler on which the event occured.

- *channelID*

  ID of the channel with the changed password.

Voice recording

A client can record all voice input and output. Client-side voice recording on all current server connections can be started and stopped with:

```
unsigned int ts3client_startVoiceRecording();
```

```
unsigned int ts3client_stopVoiceRecording();
```

When receiving or sending raw voice data, the following event is called and allows the client to handle the data, for example saving it to disk:

```
void onVoiceRecordDataEvent(data, dataSize);

const float* data;
unsigned int dataSize;
```

Parameters

- *data*

  Raw voice data buffer.

- *dataSize*

  Size of raw voice data buffer.

Miscellaneous functions

Memory dynamically allocated in the Client Lib needs to be released with:

```
unsigned int ts3client_freeMemory(pointer);

void* pointer;
```

Parameters

- *pointer*

  Address of the variable to be released.

Example:

```
char* version;

if(ts3client_getClientLibVersion(&version) == ERROR_ok) {
    printf("Version: %s\n", version);
    ts3client_freeMemory(version);
}
```

Important

Memory must not be released if the function, which dynamically allocated the memory, returned an error. In that case, the result is undefined and not initialized, so freeing the memory might crash the application.

Instead of sending the sound through the network, it can be routed directly through the playback device, so the user will get immediate audible feedback when for example configuring some sound settings.

```
unsigned int ts3client_setLocalTestMode(serverConnectionHandlerID, status);

anyID serverConnectionHandlerID;
intstatus;
```

Parameters

- *serverConnectionHandlerID*

  ID of the server connection handler for which the local test mode should be enabled or disabled.

- *status*

  Pass 1 to enable local test mode, 0 to disable.

Returns *ERROR_ok* on success, otherwise an error code as defined in public_errors.h.

FAQ

1. How to implement Push-To-Talk?

Push-To-Talk should be implemented by toggling the client variable *CLIENT_INPUT_DEACTIVATED* using the function ts3client_setClientSelfVariableAsInt. The variable can be set to the following values (see the enum InputDeactivationStatus in public_definitions.h):

- *INPUT_ACTIVE*
- *INPUT_DEACTIVATED*

For Push-To-Talk toggle between *INPUT_ACTIVE* (talking) and *INPUT_DEACTIVATED* (not talking).

Example code:

```
unsigned int error;
bool shouldTalk;

shouldTalk = isPushToTalkButtonPressed();  // Your key detection implementation
if((error = ts3client_setClientSelfVariableAsInt(scHandlerID, CLIENT_INPUT_DEACTIVATED,
                                                 shouldTalk ? INPUT_ACTIVE : INPUT_DEACTIVATED))
    != ERROR_ok) {
    char* errorMsg;
    if(ts3client_getErrorMessage(error, &errorMsg) != ERROR_ok) {
        printf("Error toggling push-to-talk: %s\n", errorMsg);
        ts3client_freeMemory(errorMsg);
    }
    return;
}

if(ts3client_flushClientSelfUpdates(scHandlerID) != ERROR_ok) {
    char* errorMsg;
    if(ts3client_getErrorMessage(error, &errorMsg) != ERROR_ok) {
        printf("Error flushing after toggling push-to-talk: %s\n", errorMsg);
        ts3client_freeMemory(errorMsg);
    }
```

```
}
```

It is not necessary to close and reopen the capture device to implement Push-To-Talk.

Basically it would be possible to toggle *CLIENT_INPUT_MUTED* as well, but the advantage of *CLIENT_INPUT_DEACTIVATED* is that the change is not propagated to the server and other connected clients, thus saving network traffic. *CLIENT_INPUT_MUTED* should instead be used for manually muting the microphone when using Voice Activity Detection instead of Push-To-Talk.

If you need to query the current muted state, use `ts3client_getClientSelfVariableAsInt`:

```
int hardwareStatus, deactivated, muted;

if(ts3client_getClientSelfVariableAsInt(scHandlerID, CLIENT_INPUT_HARDWARE,
                                        &hardwareStatus) != ERROR_ok) {
    /* Handle error */
}
if(ts3client_getClientSelfVariableAsInt(scHandlerID, CLIENT_INPUT_DEACTIVATED,
                                        &deactivated) != ERROR_ok) {
    /* Handle error */
}
if(ts3client_getClientSelfVariableAsInt(scHandlerID, CLIENT_INPUT_MUTED,
                                        &muted) != ERROR_ok) {
    /* Handle error */
}

if(hardwareStatus == HARDWAREINPUT_DISABLED) {
    /* No capture device available */
}
if(deactivated == INPUT_DEACTIVATED) {
    /* Input was deactivated for Push-To-Talk (not propagated to server) */
}
if(muted == MUTEINPUT_MUTED) {
    /* Input was muted (propagated to server) */
}
```

When using Push-To-Talk, you should deactivate Voice Activity Detection in the preprocessor or keep the VAD level very low. To deactivate VAD, use:

```
ts3client_setPreProcessorConfigValue(serverConnectionHandlerID, "vad", "false");
```

2. How to adjust the volume?

Output volume

Voice output volume can be adjusted by changing the “volume_modifier” playback option using the function `ts3client_setPlaybackConfigValue`. The value is in decibel, so 0 is no modification, negative values make the signal quieter and positive values louder.

Example to increate the output volume by 10 decibel:

```
ts3client_setPlaybackConfigValue(scHandlerID, "volume_modifier", 10);
```

Input volume

Automatic Gain Control (AGC) takes care of the input volume during preprocessing automatically. Instead of modifying the input volume directory, you modify the AGC preprocessor settings with `setProProcessorConfigValue`.

Revision history

Revision History		
Revision 1.28	29 Oct 2009	
Added ts3client_setClientVolumeModifier function to Playback chapter. Client whisper list setting is always enabled.		
Revision 1.27	05 Oct 2009	
Added port parameter to ts3client_spawnNewServerConnectionHandler and extraMessage to onServerErrorEvent		
Revision 1.26	14 Sep 2009	
Added custom encryption callbacks		
Revision 1.25	05 May 2009	
Updated documentation on getParentChannelOfChannel.		
Revision 1.24	29 Apr 2009	
Updated documentation on requestClientSetWhisperList.		
Revision 1.23	27 Mar 2009	
Renamed getCurrentPlaybackDevice/getCurrentCaptureDevice to getCurrentPlaybackDeviceName/getCurrentCaptureDeviceName. Use vacant functions to return the currently open FMOD System object.		
Revision 1.22	9 Feb 2009	
Custom FMOD objects API changes. Changed playback value voice_factor to voice_modifier, removed playWaveFile function and voice_factor_wave.		
Revision 1.21	23 Jan 2009	
Added chapter about custom FMOD objects.		
Revision 1.20	19 Dec 2008	
Added voice recording chapter.		
Revision 1.19	9 Dec 2008	
Added returnCode to functions interacting with server. Updated some functions with added uniqueIdentifier parameters.		
Revision 1.18	7 Nov 2008	
Error handling API change.		
Revision 1.17	13 Oct 2008	
Added ts3client_getServerConnectionHandlerList function.		
Revision 1.16	06 Oct 2008	
Changed function prefix from ts3_ to ts3client_ so both client and server shared libraries can be loaded in the same application.		
Revision 1.15	22 Sep 2008	
Added echo canceling to preprocessor section.		
Revision 1.14	9 Sep 2008	
Removed unused functions and enums, which were removed from the SDK headers.		
Revision 1.13	3 Sep 2008	
Removed "enabled" preprocessor flag. Changed default server port from 3000 to 9987. Adjusted ts3client_initClientLib parameters.		
Revision 1.12	8 Jul 2008	
New query clients/channel functions. New individual channel codec quality settings. Updated encoding chapter and ts3client_initClientLib() function. Removed agc_increment and agc_decrement preprocessor options.		
Revision 1.11	30 May 2008	
New server properties added. force parameter in ts3client_requestChannelDelete added. Added note about cdecl calling convention.		
Revision 1.10	22 May 2008	
Added new ts3client_createIdentity function and updated docs for onTalkStatusChangeEvent.		
Revision 1.9	16 May 2008	
Added new ts3client_getPlaybackConfigValueAsFloat() function.		
Revision 1.8	14 May 2008	
Added new mute functions and new ClientProperties fields.		
Revision 1.7	28 Apr 2008	
Added new mechanism to shutdown playback devices. Added remark about opening capture devices without closing. Added ts3client_ prefix to all clientlib function calls. Added 3D sound chapter.		
Revision 1.6	5 Mar 2008	
Added remark for new database logging.		

Index

Symbols

3D sound, 45

A

AGC, 39
architecture, 5
Automatic Gain Control, 39

B

bandwidth, 37

C

callback, 10
calling convention, 7
capture device, 24
Channel order, 65
client ID, 17
codec, 37
contact, 2
copyright, 2

E

echo canceling, 39
encoder, 38
enums
  ChannelProperties, 60
  ClientProperties, 52, 56
  ConnectStatus, 16, 19, 66
  InputDeactivationStatus, 91
  LogLevel, 21, 22
  LogType, 10, 22
  TextMessageTargetMode, 77, 78
  VirtualServerProperties, 67
  Visibility, 71, 81, 83
error codes, 8
events
  onChannelDescriptionUpdateEvent, 88
  onChannelMoveEvent, 76
  onChannelPasswordChangedEvent, 89
  onChannelSubscribeEvent, 82
  onChannelSubscribeFinishedEvent, 82
  onChannelUnsubscribeEvent, 83
  onChannelUnsubscribeFinishedEvent, 83
  onClientKickFromChannelEvent, 80
  onClientKickFromServerEvent, 80
  onClientMoveEvent, 70
  onClientMoveMovedEvent, 72

onClientMoveSubscriptionEvent, 83
onClientMoveTimeoutEvent, 87
onConnectStatusChangeEvent, 16, 18
onCustomCaptureDeviceCloseEvent, 35
onCustomPacketDecryptEvent, 86
onCustomPacketEncryptEvent, 85
onCustomPlaybackDeviceCloseEvent, 35
onDelChannelEvent, 75
onFMODChannelCreatedEvent, 35
onNewChannelCreatedEvent, 73
onNewChannelEvent, 17
onPlaybackShutdownCompleteEvent, 31
onServerEditedEvent, 69
onServerErrorEvent, 9, 21
onServerStopEvent, 19
onServerUpdatedEvent, 68
onTalkStatusChangeEvent, 87
onTextMessageEvent, 78
onUpdateChannelEditedEvent, 64
onUpdateChannelEvent, 89
onUpdateClientEvent, 58
onUserLoggingMessageEvent, 22
onVoiceRecordDataEvent, 90

F

FAQ, 91
functions
ts3client_activateCaptureDevice, 32
ts3client_activateCustomCaptureDevice, 33
ts3client_closeCaptureDevice, 30
ts3client_closePlaybackDevice, 30
ts3client_createIdentity, 13
ts3client_destroyClientLib, 12
ts3client_destroyServerConnectionHandler, 13
ts3client_flushChannelCreation, 72
ts3client_flushChannelUpdates, 63
ts3client_flushClientSelfUpdates, 55
ts3client_fmod_Channelset3DAttributes, 47
ts3client_fmod_Systemset3DListenerAttributes, 45
ts3client_fmod_Systemset3DSettings, 46
ts3client_freeMemory, 90
ts3client_getCaptureDeviceList, 28
ts3client_getCaptureModeList, 26
ts3client_getChannelClientList, 49
ts3client_getChannelIDFromChannelNames, 65
ts3client_getChannelList, 48
ts3client_getChannelOfClient, 49
ts3client_getChannelVariableAsInt, 60
ts3client_getChannelVariableAsString, 60
ts3client_getClientID, 17, 51
ts3client_getClientLibVersion, 11
ts3client_getClientList, 48

ts3client_getClientSelfVariableAsInt, 52
ts3client_getClientSelfVariableAsString, 52
ts3client_getClientVariableAsInt, 57
ts3client_getClientVariableAsString, 57
ts3client_getConnectionStatus, 16
ts3client_getCurrentCaptureDevice, 36
ts3client_getCurrentCaptureDeviceName, 29
ts3client_getCurrentCaptureMode, 29
ts3client_getCurrentPlaybackDevice, 36
ts3client_getCurrentPlaybackDeviceName, 29
ts3client_getCurrentPlayBackMode, 29
ts3client_getDefaultCaptureDevice, 27
ts3client_getDefaultCaptureMode, 25
ts3client_getDefaultPlaybackDevice, 27
ts3client_getDefaultPlayBackMode, 25
ts3client_getEncodeConfigValue, 38
ts3client_getErrorMessage, 20
ts3client_getParentChannelOfChannel, 50
ts3client_getPlaybackConfigValueAsFloat, 42
ts3client_getPlaybackDeviceList, 28
ts3client_getPlaybackModeList, 26
ts3client_getPreProcessorConfigValue, 39
ts3client_getPreProcessorInfoValueFloat, 42
ts3client_getServerConnectionHandlerList, 47
ts3client_getServerVariableAsInt, 66
ts3client_getServerVariableAsString, 66
ts3client_getServerVariableAsUInt64, 66
ts3client_initClientLib, 9
ts3client_initiateGracefulPlaybackShutdown, 30
ts3client_logMessage, 21
ts3client_openCaptureDevice, 24
ts3client_openCustomCaptureDevice, 33
ts3client_openCustomPlaybackDevice, 32
ts3client_openPlaybackDevice, 24
ts3client_requestChannelDelete, 74
ts3client_requestChannelDescription, 61
ts3client_requestChannelMove, 75
ts3client_requestChannelSubscribe, 81
ts3client_requestChannelSubscribeAll, 82
ts3client_requestChannelUnsubscribe, 82
ts3client_requestChannelUnsubscribeAll, 82
ts3client_requestClientKickFromChannel, 79
ts3client_requestClientKickFromServer, 79
ts3client_requestClientMove, 70
ts3client_requestClientSetWhisperList, 59
ts3client_requestClientVariables, 58
ts3client_requestMuteClients, 84
ts3client_requestSendTextMsg, 77
ts3client_requestServerVariables, 68
ts3client_requestUnmuteClients, 84
ts3client_setChannelVariableAsInt, 62
ts3client_setChannelVariableAsString, 63

ts3client_setClientSelfVariableAsInt, 55
ts3client_setClientSelfVariableAsString, 55
ts3client_setClientVolumeModifier, 44
ts3client_setLocalTestMode, 91
ts3client_setLogVerbosity, 23
ts3client_setPlaybackConfigValue, 43, 92
ts3client_setPreProcessorConfigValue, 40
ts3client_spawnNewServerConnectionHandler, 12
ts3client_startConnection, 14
ts3client_startVoiceRecording, 89
ts3client_stopConnection, 18
ts3client_stopVoiceRecording, 89

H

headers, 7

L

Linux, 7
Logging, 21

M

Macintosh, 7

N

narrowband, 37

P

Permanent channel, 62
playback device, 24
preprocessor, 39
PushToTalk, 91

R

return code, 9

S

sampling rates, 37
Semi-permanent channel, 62
server connection handler, 12
structs
TS3CLIENT_FMOD_VECTOR, 45
system requirements, 7

T

TeamSpeak Systems, 2

U

ultra-wideband, 37

V

VAD, 39

Voice Activity Detection, 39
volume_modifier, 43, 92

W

welcome message, 16
wideband, 37
Windows, 7