TeamSpeak 3 Client
SDK Developer Manual

Revision 2017-09-08 09:51:42

Table of Contents

Introduction

TeamSpeak 3 is 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.

The cross-platform Software Development Kit allows the easy integration of the TeamSpeak client and server technology into own applications.

Tis 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.

System requirements

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

- Windows

  Windows XP, Vista, Windows 7, 8, 8.1 (32- and 64-bit)

- Mac OS X

  Mac OS X 10.6 and above

- Linux

  Any recent Linux distribution with libstdc++ 6 (32- and 64-bit)

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_sdk_definitions.h`

Enum definitions for filetransfer support.

- `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(uint64 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, resourcesFolder);

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

- *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

LogType_NO_NETLOGGING is no longer used. Previously this controlled if the Client Lib would send warning, error and critical log entries to a webserver for analysis. As netlogging does not occur anymore, this flag has no effect anymore.

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

- *logFileFolder*

  If file logging is used, this defines the location where the logs are written to. Pass NULL for the default behaviour, which is to use a folder called `logs` in the current working directory.

  resourcesFolder

  Resource path pointing to the directory where the soundbackends folder is located. Required so your application finds the sound backend shared libraries. This should usually point to the root or bin directory of your application, depending where the soundbackends directory is located.

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.

Note

All the clientlib callbacks are asynchronous, except for the sound callbacks which allow to directly manipulate the sound buffer.

A callback example in C:

```
static void my_onConnectStatusChangeEvent_Callback(uint64 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 structs to avoid polluting the SDK headers with code used only internally.

Querying the library version

The complete Client Lib version string can be queried with

`unsigned int` **`ts3client_getClientLibVersion`**`(`*`result`*`);`

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

- *`result`*

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

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.

To get only the version number, which is a part of the complete version string, as numeric value:

`unsigned int` **`ts3client_getClientLibVersionNumber`**`(`*`result`*`);`

`uint64*` *`result`*`;`

- *result*

  Address of a variable that receives the numeric clientlib version.

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

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 */
```

Example using ts3client_getClientLibVersionNumber:

```
unsigned int error;
uint64 version;
error = ts3client_getClientLibVersionNumber(&version);
if(error != ERROR_ok) {
    printf("Error querying clientlib version number: %d\n", error);
    return;
}
printf("Client library version number: %ld\n", version);  /* Print version */
```

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;
uint64* result;
```

- *port*

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

  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 need a specific port. Passing zero is the better way in most use cases.

- *result*

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

To destroy a server connection handler, call

```
unsigned int ts3client_destroyServerConnectionHandler(serverConnectionHandlerID);

uint64 serverConnectionHandlerID;
```

- *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;
```

- *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);

uint64 serverConnectionHandlerID;
const char* identity;
const char* ip;
unsigned int port;
const char* nickname;
const char** defaultChannelArray;
const char* defaultChannelPassword;
const char* serverPassword;
```

- *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*.

There is an alternative convinience function to start the connection which takes a channelID as parameter for the default channel instead of a channel name string array.

```
unsigned int ts3client_startConnectionWithChannelID(serverConnectionHandlerID, identity, ip, port, nickname, defaultChannelId, defaultChannelPassword, serverPassword);
```

```
uint64 serverConnectionHandlerID;
const char* identity;
const char* ip;
unsigned int port;
const char* nickname;
uint64 defaultChannelId;
const char* defaultChannelPassword;
const char* serverPassword;
```

- *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.

- *`defaultChannelID`*

  Specifies ID of the channel on the TeamSpeak server we want to connect to. This is an alternative way to define the channel by ID instead of channel path as in `ts3client_startConnection`. If the specified channel does no longer exist or if the channel password is incorrect, the user will be connected to the default channel of the TeamSpeak server.

- *`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.

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

```
unsigned int error;
uint64 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);

uint64 serverConnectionHandlerID;
int newStatus;
int errorNumber;
```

- *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);
```

```
uint64 serverConnectionHandlerID;
int* result;
```

- *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);
```

```
uint64 serverConnectionHandlerID;
anyID* result;
```

- *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);
```

```
uint64 serverConnectionHandlerID;
uint64 channelID;
uint64 channelParentID;
```

- *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);

uint64 serverConnectionHandlerID;
const char* quitMessage;
```

- *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(serverConnectionHandlerID, newStatus, errorNumber);

uint64 serverConnectionHandlerID;
int newStatus;
int errorNumber;
```

- *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);

uint64 serverConnectionHandlerID;
const char* shutdownMessage;
```

- *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) {
    /* Use welcomeMsg... */
    ts3client_freeMemory(welcomeMsg);  /* Release memory *only* if function did not return an error */
} else {
    /* Handle error */
    /* Do not access or release welcomeMessage, the variable is undefined */
}
```

Important

Client Lib functions returning C-strings or arrays 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;
```

- *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 */
    }
}
```

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);

uint64 serverConnectionHandlerID;
const char* errorMessage;
unsigned int error;
const char* returnCode;
const char* extraMessage;
```

- *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;
uint64 logID;
```

- *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;
uint64 logID;
const char* logTime;
const char* completeLogString;
```

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

- *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;
```

- *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. Accessing devices is handled by the sound backend shared libraries, found in the soundbackends directory in the SDK. There are different backends available on the supported operating systems: DirectSound and Windows Audio Session API on Windows, Alsa and PulseAudio on Linux, CoreAudio on Mac OS X.

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);

uint64 serverConnectionHandlerID;
const char* modeID;
const char* playbackDevice;
```

- *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.

  Passing an empty string will use the default playback mode.

- *playbackDevice*

  Valid parameters are:

  - The *device* parameter returned by ts3client_getDefaultPlaybackDevice
  - One of the *device* parameters returned by ts3client_getPlaybackDeviceList

- Empty string to initialize the default playback device.
- Linux with Alsa only: Custom device name in the form of e.g. “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 the sound backend fails to find a usable playback device.

```
unsigned int ts3client_openCaptureDevice(serverConnectionHandlerID, modeID, capture-
Device);

uint64 serverConnectionHandlerID;
const char* modeID;
const char* captureDevice;
```

- *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.

  Passing an empty string will use the default capture mode.

- *captureDevice*

  Valid parameters are:

  - The *device* parameter returned by ts3client_getDefaultCaptureDevice
  - One of the *device* parameters returned by ts3client_getCaptureDeviceList
  - Empty string to initialize the default capture device. Encoded in UTF-8 format.
  - Linux with Alsa only: Custom device name in the form of e.g. “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: DirectSound on all Windows platforms, Windows Audio Session API for Windows Vista and Windows 7; Alsa and PulseAudio on Linux; CoreAudio on Mac OS X.

Available device names may differ depending on the current mode.

The default playback and capture modes can be queried with:

```
unsigned int ts3client_getDefaultPlayBackMode(result);

char** result;
```

```
unsigned int ts3client_getDefaultCaptureMode(result);

char** result;
```

- *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. 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`.

All available playback and capture modes can be queried with:

```
unsigned int ts3client_getPlaybackModeList(result);

char*** result;
```

```
unsigned int ts3client_getCaptureModeList(result);

char*** result;
```

- *result*

  Address of a variable that receives a NULL-terminated array of C-strings listing available playback or capture modes.

  Unless the function returns an error, the caller must release each element of the array (the C-string) and finally the complete array with `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 accessed or released.

Example to query all available playback modes:

```
char** array;

if(ts3client_getPlaybackModeList(&array) == ERROR_ok) {
    for(int i=0; array[i] != NULL; ++i) {
        printf("Mode: %s\n", array[i]);
        ts3client_freeMemory(array[i]);  // Free C-string
    }
    ts3client_freeMemory(array);  // Free the array
}
```

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

To query the default playback and capture device, call

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

const char* modeID;
char*** result;


unsigned int ts3client_getDefaultCaptureDevice(modeID, result);

const char* modeID;
char*** result;
```

- *mode*

  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 of two C-strings. The first element contains the device name, the second the device ID.

  Unless the function returns an error, the caller must free the two array elements and the complete array with `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:

```
char* defaultMode;

/* Get default playback mode */
if(ts3client_getDefaultPlayBackMode(&defaultMode) == ERROR_ok) {
    char** defaultPlaybackDevice;

    /* Get default playback device */
    if(ts3client_getDefaultPlaybackDevice(defaultMode, &defaultPlaybackDevice) == ERROR_ok) {
        printf("Default playback device name: %s\n", defaultPlaybackDevice[0]);  /* First element: Device name */
        printf("Default playback device ID: %s\n",   defaultPlaybackDevice[1]);  /* Second element: Device ID */

        /* Release the two array elements and the array */
        ts3client_freeMemory(defaultPlaybackDevice[0]);
        ts3client_freeMemory(defaultPlaybackDevice[1]);
        ts3client_freeMemory(defaultPlaybackDevice);
    } else {
        printf("Failed to get default playback device\n");
    }
} else {
    printf("Failed to get default playback mode\n");
}
```

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

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

const char* modeID;
char**** result;
```

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

const char* modeID;
char**** result;
```

- *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* deviceName, char* deviceID }, { char* deviceName, char* deviceID }, ... , NULL }.

  Unless the function returns an error, the elements of the array and the array itself need 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:

```
char* defaultMode;

if(ts3client_getDefaultPlayBackMode(&defaultMode) == ERROR_ok) {
    char*** array;

    if(ts3client_getPlaybackDeviceList(defaultMode, &array) == ERROR_ok) {
        for(int i=0; array[i] != NULL; ++i) {
            printf("Playback device name: %s\n", array[i][0]);  /* First element: Device name */
            printf("Playback device ID: %s\n",   array[i][1]);  /* Second element: Device ID */

            /* Free element */
            ts3client_freeMemory(array[i][0]);
            ts3client_freeMemory(array[i][1]);
            ts3client_freeMemory(array[i]);
        }
        ts3client_freeMemory(array);  /* Free complete array */
    } else {
        printf("Error getting playback device list\n");
    }
} else {
    printf("Error getting default playback mode\n");
}
```

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);

uint64 serverConnectionHandlerID;
char** result;
```

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

uint64 serverConnectionHandlerID;
char** result;
```

- *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. 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`.

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

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

uint64 serverConnectionHandlerID;
char** result;
int* isDefault;
```

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

uint64 serverConnectionHandlerID;
char** result;
int* isDefault;
```

- *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`.

- *`result`*

  Address of a variable that receives a flag if this device is the default playback/capture device. If this is not needed, pass NULL instead.

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);

uint64 serverConnectionHandlerID;


unsigned int ts3client_closePlaybackDevice(serverConnectionHandlerID);

uint64 serverConnectionHandlerID;
```

- *`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);

uint64 serverConnectionHandlerID;
```

- *`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);

uint64 serverConnectionHandlerID;
```

- *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(uint64 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.

Using custom devices

Instead of opening existing sound devices that TeamSpeak has detected, you can also use our custom capture and playback mechanism to allow you to override the way in which TeamSpeak does capture and playback. When you have opened a custom capture and playback device you must regularly supply new "captured" sound data via the `ts3client_processCustomCaptureData` function and retrieve data that should be "played back" via `ts3client_acquireCustomPlaybackData`. Where exactly this captured sound data comes from and where the playback data goes to is up to you, which allows a lot of cool things to be done with this mechanism.

Implementing own custom devices is for special use cases and entirely optional.

Registering a custom device announces the device ID and name to the Client Lib. Once a custom device has been registered under a device ID, the device can be opened like any standard device with `ts3client_openCaptureDevice` and `ts3client_openPlaybackDevice`.

```
void ts3client_registerCustomDevice(deviceID, deviceDisplayName, capFrequency,
capChannels, playFrequency, playChannels);

const char* deviceID;
const char* deviceDisplayName;
int capFrequency;
int capChannels;
```

```
int playFrequency;
int playChannels;
```

- *deviceID*

  ID string of the custom device, under which the device can be later accessed.

- *deviceDisplayName*

  Displayed name of the custom device. Freely choose a name which identifies your device.

- *capFrequency*

  Frequency of the capture device.

- *capChannels*

  Number of channels of the capture device. This value depends on if the used codec is a mono or stereo codec.

- *playFrequency*

  Frequency of the playback device.

- *playChannels*

  Number of channels of the playback device.

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

Unregistering a custom device will automatically close the device:

```
void ts3client_unregisterCustomDevice(deviceID);

const char* deviceID;
```

- *deviceID*

  ID string of the custom device to unregister. This is the ID under which the device was registered with ts3client_registerCustomDevice.

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

To send the captured data from your device to the Client Lib:

```
void ts3client_processCustomCaptureData(deviceID, buffer, samples);

const char* deviceID;
const short* buffer;
int samples;
```

- *deviceID*

ID string of the custom device. This is the ID under which the device was registered with `ts3client_registerCustomDevice`.

- *`buffer`*

  Capture data buffer containing the data captured by the custom device.

- *`samples`*

  Size of the capture data buffer.

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

Retrieve playback data from the Client Lib:

```
void ts3client_acquireCustomPlaybackData(deviceID, buffer, samples);
```

```
const char* deviceID;
const short* buffer;
int samples;
```

- *`deviceID`*

  ID string of the custom device. This is the ID under which the device was registered with `ts3client_registerCustomDevice`.

- *`buffer`*

  Buffer containing the playback data retrieved from the Client Lib.

- *`samples`*

  Size of the playback data buffer.

Returns *`ERROR_ok`* if playback data is available or *`ERROR_sound_no_data`* if the Client Lib currently has no playback data.

The return value *`ERROR_sound_no_data`* can be used for performance optimisation, it means there is currently only silence (nobody is talking, no wave files being played etc.) and instead of returning a buffer full of zeroes it just notifies the user there is currently no data, which allows you to not playback any sound data for that moment, if your API supports that (potentially saving some CPU), or to just fill the sound buffer with zeroes and playback this if your sound API demands you to fill it with something for every given time.

Overview on registering and opening a custom device:

```
/* Register a new custom sound device with specified frequency and number of channels */
if(ts3client_registerCustomDevice("customWaveDeviceId", "Nice displayable wave device name", captureFrequency, captu
    printf("Failed to register custom device\n");
}

/* Open capture device we created earlier */
if(ts3client_openCaptureDevice(scHandlerID, "custom", "customWaveDeviceId") != ERROR_ok) {
```

```
    printf("Error opening capture device\n");
}

/* Open playback device we created earlier */
if(ts3client_openPlaybackDevice(scHandlerID, "custom", "customWaveDeviceId") != ERROR_ok) {
    printf("Error opening playback device\n");
}

/* Main loop */
while(!abort) {
    /* Fill captureBuffer from your custom device */

    /* Stream your capture data to the client lib */
 if(ts3client_processCustomCaptureData("customWaveDeviceId", captureBuffer, captureBufferSize) != ERROR_ok) {
        printf("Failed to process capture data\n");
    }

    /* Get playback data from the client lib */
    error = ts3client_acquireCustomPlaybackData("customWaveDeviceId", playbackBuffer, playbackBufferSize);
    if(error == ERROR_ok) {
        /* Playback data available, send playbackBuffer to your custom device */
    } else if(error == ERROR_sound_no_data) {
        /* Not an error. The client lib has no playback data available. Depending on your custom sound API, either
           pause playback for performance optimisation or send a buffer of zeros. */
    } else {
        printf("Failed to get playback data\n");  /* Error occured */
    }
}

/* Unregister the custom device. This automatically close the device. */
if(ts3client_unregisterCustomDevice("customaveDeviceId") != ERROR_ok) {
    printf("Failed to unregister custom device\n");
}
```

Note

Further sample code on how to use a custom device can be found in the “client_customdevice” example included in the SDK.

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`*`);`

`uint64` *`serverConnectionHandlerID`*`;`

- *`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`.

Sound codecs

TeamSpeak 3 supports the following sound sampling rates:

- Speex Narrowband (8 kHz)
- Speex Wideband (16 kHz)
- Speex Ultra-Wideband (32 kHz)
- Celt (Mono, 48kHz)
- Opus Voice (Mono, 48khz)
- Opus Music (Stereo, 48khz)

Note

Opus Voice is recommended for voice transmission. Speex and Celt codecs may be removed in future versions of this SDK.

Bandwidth usage generally depends on the used codec and the encoders quality setting.

Estimated bitrates (bps) for codecs per quality:

Quality	Narrowband	Wideband	Ultra-Wide-band	Celt	Opus Voice	Opus Music
0	2,150	3,950	4,150	32,000	4,096	7,200
1	3,950	5,750	7,550	32,000	8,192	14,400
2	5,950	7,750	9,550	40,000	12,288	21,600
3	8,000	9,800	11,600	40,000	16,384	28,800
4	8,000	12,800	14,600	40,000	20,480	36,000
5	11,000	16,800	18,600	48,000	24,576	43,200
6	11,000	20,600	22,400	48,000	28,672	50,400
7	15,000	23,800	25,600	48,000	32,768	57,600
8	15,000	27,800	29,600	48,000	36,864	64,800
9	18,200	34,400	36,200	64,000	40,960	72,000
10	24,600	42,400	44,200	96,000	45,056	79,200

Change the quality to find a good middle between voice quality and bandwidth usage. Overall the Opus codec delivers the best quality per used bandwidth.

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);
```

Available values for *`CHANNEL_CODEC`* are:

- 0 - Speex Narrowband)
- 1 - Speex Wideband
- 2 - Speex Ultra-Wideband
- 3 - Celt
- 4 - Opus Voice
- 5 - Opus Music

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);
```

```
uint64 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) and voice activity detection (VAD).

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);

uint64 serverConnectionHandlerID;
const char* ident;
char** result;
```

- *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"

    Checks if echo canceling is enabled. Returns a string with a boolean value.

- *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);
```

```
uint64 serverConnectionHandlerID;
const char* ident;
```

```
const char* value;
```

- *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 echo canceling. Boolean value converted to string. Default is "false".

- *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);

uint64 serverConnectionHandlerID;
const char* ident;
float* result;
```

- *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);

uint64 serverConnectionHandlerID;
const char* ident;
float* result;
```

- *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.

- “volume_factor_wave”

  Adjust the volume of wave files played by `ts3client_playWaveFile` and `ts3client_playWaveFileHandle`. The value is a float defining the volume reduction in decibel. Reasonable values range from “-40.0” (very silent) to “0.0” (loudest).

- *`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);
```

```
uint64 serverConnectionHandlerID;
const char* ident;
const char* value;
```

- *`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);

uint64 serverConnectionHandlerID;
anyID clientID;
float value;
```

- *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;
}
```

Accessing the voice buffer

The TeamSpeak Client Lib allows users to access the raw playback and capture voice data and even modify it, for example to add effects to the voice. These callbacks are also used by the TeamSpeak client for the voice recording feature.

Using these low-level callbacks is not required and should be reserved for specific needs. Most SDK applications won't need to implement these callbacks.

The following event is called when a voice packet from a client (not own client) is decoded and about to be played over your sound device, but before it is 3D positioned and mixed with other sounds. You can use this function to alter the voice data (for example when you want to do effects on it) or to simply get voice data. The TeamSpeak client uses this function to record sessions.

```
void onEditPlaybackVoiceDataEvent(serverConnectionHandlerID, clientID, samples, sam-
pleCount, channels);

uint64 serverConnectionHandlerID;
anyID clientID;
short* samples;
int sampleCount;
int channels;
```

- *serverConnectionHandlerID*

  ID of the server connection handler from which the voice data was sent.

- *clientID*

  ID of the client whose voice data is received.

- *samples*

  Pointer to the voice data (signed 16 bit @ 48KHz).

- *sampleCount*

  Number of samples the "samples" variable points to.

- *channels*

  Number of channels in the sound data.

The following event is called when a voice packet from a client (not own client) is decoded and 3D positioned and about to be played over your sound device, but before it is mixed with other sounds. You can use this function to alter/get the voice data after 3D positioning.

```
void onEditPostProcessVoiceDataEvent(serverConnectionHandlerID, clientID, samples,
sampleCount, channels, channelSpeakerArray, channelFillMask);

uint64 serverConnectionHandlerID;
anyID clientID;
short* samples;
int sampleCount;
int channels;
const unsigned int* channelSpeakerArray;
unsigned int* channelFillMask;
```

- *serverConnectionHandlerID*

  ID of the server connection handler from which the voice data was sent.

- *clientID*

  ID of the client whose voice data is received.

- *samples*

  Pointer to the voice data (signed 16 bit @ 48KHz).

- *sampleCount*

  Number of samples the "samples" variable points to.

- *channels*

  Number of channels in the sound data.

- *channelSpeakerArray*

  An array with *channels* entries, defining the speaker each channels represents. The speaker values can be found in the SPEAKER_* defines within public_definitions.h.

  For example for stereo (*channels* = 2), the array might look liks this:

  ```
  channelSpeakerArray[0] = SPEAKER_FRONT_LEFT
  channelSpeakerArray[1] = SPEAKER_FRONT_RIGHT
  ```

- *channelFillMask*

  A pointer to a bit-mask defining which channels are filled. For efficiency reasons, not all channels need to have actual sound data in it. So before this data is used, use this bit-mask to check if the channel is actually filled. If you decide to add data to a channel that is empty, set the bit for this channel in this mask.

For example, this callback reports:

channels = 6
channelSpeakerArray[0] = SPEAKER_FRONT_CENTER
channelSpeakerArray[1] = SPEAKER_LOW_FREQUENCY
channelSpeakerArray[2] = SPEAKER_BACK_LEFT
channelSpeakerArray[3] = SPEAKER_BACK_RIGHT
channelSpeakerArray[4] = SPEAKER_SIDE_LEFT

channelSpeakerArray[5] = SPEAKER_SIDE_RIGHT // Quite unusual setup
*channelFillMask = 1

This means "samples" points to 6 channel data, but only the SPEAKER_FRONT_CENTER channel has data, the other channels are undefined (not necessarily 0, but undefined).

So for the first sample, samples[0] has data and samples[1], samples[2], samples[3], samples[4] and samples[5] are undefined.

If you want to add SPEAKER_BACK_RIGHT channel data you would do something like:

```
*channelFillMask |= 1<<3;  // SPEAKER_BACK_RIGHT is the 4th channel (is index 3) according to *channelSpeakerArray.
for(int i=0; i<sampleCount; ++i){
    samples[3 + (i*channels) ] = getChannelSoundData(SPEAKER_BACK_RIGHT, i);
}
```

The following event is called when all sounds that are about to be played back for this server connection are mixed. This is the last chance to alter/get sound.

You can use this function to alter/get the sound data before playback.

```
void onEditMixedPlaybackVoiceDataEvent(serverConnectionHandlerID, samples, sampleCount, channels, channelSpeakerArray, channelFillMask);
```

```
uint64 serverConnectionHandlerID;
short* samples;
int sampleCount;
int channels;
const unsigned int* channelSpeakerArray;
unsigned int* channelFillMask;
```

- *serverConnectionHandlerID*

  ID of the server connection handler from which the voice data was sent.

- *samples*

  Pointer to the voice data (signed 16 bit @ 48KHz).

- *sampleCount*

  Number of samples the "samples" variable points to.

- *channels*

  Number of channels in the sound data.

- *channelSpeakerArray*

  An array with *channels* entries, defining the speaker each channels represents. The speaker values can be found in the SPEAKER_* defines within public_definitions.h.

  For example for stereo (*channels* = 2), the array might look liks this:

  ```
  channelSpeakerArray[0] = SPEAKER_FRONT_LEFT
  channelSpeakerArray[1] = SPEAKER_FRONT_RIGHT
  ```

- *channelFillMask*

  A pointer to a bit-mask of which channels are filled. For efficiency reasons, not all channels need to have actual sound data in it. So before this data is used, use this bit-mask to check if the channel is actually filled. If you decide to add data to a channel that is empty, set the bit for this channel in this mask.

The following event is called after sound is recorded from the sound device and is preprocessed. This event can be used to get/alter recorded sound. Also it can be determined if this sound will be send, or muted. This is used by the TeamSpeak client to record sessions.

If the sound data will be send, (*edited | 2) is true. If the sound data is changed, set bit 1 (*edited |=1). If the sound should not be send, clear bit 2. (*edited &= ~2)

```
void onEditCapturedVoiceDataEvent(serverConnectionHandlerID, samples, sampleCount,
channels, edited);
```

```
uint64 serverConnectionHandlerID;
short* samples;
int sampleCount;
int channels;
int* edited;
```

- *serverConnectionHandlerID*

  ID of the server connection handler from which the voice data was sent.

- *samples*

  Pointer to the voice data (signed 16 bit @ 48KHz).

- *sampleCount*

  Number of samples the "samples" variable points to.

- *channels*

  Number of channels in the sound data.

- *edited*

  When called, bit 2 indicates if the sound is about to be sent to the server.

  On return, set bit 1 if the sound data was changed.

Voice recording

When using the above callbacks to record voice, you should notify the server when recording starts or stops with the following functions:

```
unsigned int ts3client_startVoiceRecording(serverConnectionHandlerID);
```

```
uint64 serverConnectionHandlerID;
```

```
unsigned int ts3client_stopVoiceRecording(serverConnectionHandlerID);

uint64 serverConnectionHandlerID;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which voice recording should be started or stopped.

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

Playing wave files

The TeamSpeak Client Lib offers support to play wave files from the local harddisk.

To play a local wave file, call

```
unsigned int ts3client_playWaveFile(serverConnectionHandlerID, path);

anyID serverConnectionHandlerID;
const char* path;
```

- *serverConnectionHandlerID*

  ID of the server connection handler defining which playback device is to be used to play the sound file.

- *path*

  Local filepath of the sound file in WAV format to be played, encoded in UTF-8.

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

This is the simple version of playing a sound file. It's a fire-and-forget mechanism, this function will not block.

The more complex version is to play an optionally looping sound and obtain a handle, which can be used to pause, unpause and stop the loop.

```
unsigned int ts3client_playWaveFileHandle(serverConnectionHandlerID, path, loop,
waveHandle);

anyID serverConnectionHandlerID;
const char* path;
int loop;
uint64* waveHandle;
```

- *serverConnectionHandlerID*

  ID of the server connection handler defining which playback device is to be used to play the sound file.

- *path*

  Local filepath of the sound file in WAV format to be played, encoded in UTF-8.

- *loop*

  If set to 1, the sound will be looping until the handle is paused or closed.

- *waveHandle*

  Memory address of a variable in which the handle is written. Use this handle to call `ts3client_pauseWaveFileHandle` and `ts3client_closeWaveFileHandle`.

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

Using the handle obtained by `ts3client_playWaveFileHandle`, sounds can be paused and unpaused with

```
unsigned int ts3client_pauseWaveFileHandle(serverConnectionHandlerID, waveHandle, pause);
```

```
anyID serverConnectionHandlerID;
uint64 waveHandle;
int pause;
```

- *serverConnectionHandlerID*

  ID of the server connection handler defining which playback device is to be used to play the sound file.

- *waveHandle*

  Wave handle obtained by `ts3client_playWaveFileHandle`.

- *pause*

  If set to 1, the sound will be paused. Set to 0 to unpause.

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

Using the handle obtained by `ts3client_playWaveFileHandle`, sounds can be closed with

```
unsigned int ts3client_closeWaveFileHandle(serverConnectionHandlerID, waveHandle);
```

```
anyID serverConnectionHandlerID;
uint64 waveHandle;
```

- *serverConnectionHandlerID*

  ID of the server connection handler defining which playback device is to be used to play the sound file.

- *waveHandle*

  Wave handle obtained by ts3client_playWaveFileHandle.

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

3D Sound

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

Generally the struct TS3_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. */
} TS3_VECTOR;
```

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

unsigned int **ts3client_systemset3DListenerAttributes**(*serverConnectionHandlerID*, *position*, *forward*, *up*);

```
uint64 serverConnectionHandlerID;
const TS3_VECTOR* position;
const TS3_VECTOR* forward;
const TS3_VECTOR* up;
```

- *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.

- *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.

To adjust 3D sound system settings use:

```
unsigned int ts3client_systemset3DSettings(serverConnectionHandlerID, distanceFactor, rolloffScale);

uint64 serverConnectionHandlerID;
float distanceFactor;
float rolloffScale;
```

- *serverConnectionHandlerID*

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

- *distanceFactor*

  Relative distance factor. Default is 1.0 = 1 meter

- *rolloffScale*

  Scaling factor for 3D sound rolloff. Defines how fast sound volume will attenuate. As higher the value, as faster the sound is toned with increasing distance.

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

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

```
unsigned int ts3client_channelset3DAttributes(serverConnectionHandlerID, clientID, position);

uint64 serverConnectionHandlerID;
anyID clientID;
const TS3_VECTOR* position;
```

- *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.

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

The following event is called to calculate volume attenuation for distance in 3D positioning of clients.

```
void onCustom3dRolloffCalculationClientEvent(serverConnectionHandlerID, clientID,
distance, volume);

uint64 serverConnectionHandlerID;
anyID clientID;
float distance;
float* volume;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which the volume attenuation calculation occured.

- *clientID*

  ID of the client which is being 3D positioned.

- *distance*

  The distance between the listener and the client.

- *volume*

  The volume which the Client Lib calculated. This can be changed in this callback.

The following event is called to calculate volume attenuation for distance in 3D positioning of a wave file that was opened previously with `ts3client_playWaveFileHandle`.

```
void onCustom3dRolloffCalculationWaveEvent(serverConnectionHandlerID, waveHandle,
distance, volume);

uint64 serverConnectionHandlerID;
uint64 waveHandle;
float distance;
float* volume;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which the volume attenuation calculation occured.

- *waveHandle*

  Handle for the playing wave file, returned by `ts3client_playWaveFileHandle`.

- *distance*

  The distance between the listener and the client.

- *volume*

  The volume which the Client Lib calculated. This can be changed in this callback.

This method is used to 3D position a wave file that was opened previously with `ts3client_playWaveFileHandle`.

```
unsigned int ts3client_set3DWaveAttributes(serverConnectionHandlerID, waveHandle, position);

uint64 serverConnectionHandlerID;
uint64 waveHandle;
const TS3_VECTOR* position;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which the volume attenuation calculation occured.

- *waveHandle*

  Handle for the playing wave file, returned by `ts3client_playWaveFileHandle`.

- *position*

  The 3D position of the sound.

- *volume*

  The volume which the Client Lib calculated. This can be changed in this callback.

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

Query available servers, channels and clients

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

```
unsigned int ts3client_getServerConnectionHandlerList(result);

uint64** result;
```

- *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);

uint64 serverConnectionHandlerID;
uint64** result;
```

- *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);
```

```
uint64 serverConnectionHandlerID;
anyID** result;
```

- *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 if the channel is currently subscribed:

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

```
uint64 serverConnectionHandlerID;
uint64 channelID;
anyID** result;
```

- *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(serverConnectionHandlerID, clientID, result);

uint64 serverConnectionHandlerID;
anyID clientID;
uint64* result;
```

- *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(serverConnectionHandlerID, channelID, result);

uint64 serverConnectionHandlerID;
uint64 channelID;
uint64* result;
```

- *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:

```
uint64* 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:

```
uint64 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);

uint64 serverConnectionHandlerID;
anyID* result;
```

- *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);

uint64 serverConnectionHandlerID;
ClientProperties flag;
int* result;
```

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

uint64 serverConnectionHandlerID;
ClientProperties flag;
char** result;
```

- *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_OUTPUTONLY_MUTED         //automatically up-to-date for any client "in view", this clients
                                  //headphones/speakers only 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_VERSION_SIGN,     //internal use
  CLIENT_SECURITY_HASH,           //SDK only: Hash is provided by an outside source. A channel will
                                  //use the security salt + other client data to calculate a hash,
          //which must be the same as the one provided here.
  CLIENT_ENCRYPTION_CIPHERS,      //SDK only: list of available ciphers send to the server
  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 users.

- *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.

Note: You should query this flag for the own client using `ts3client_getClientSelfVariableAsInt`.

- *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 combined mute status of the clients playback and capture devices. Possible values are defined by the enum MuteOutputStatus. Always available for visible clients.

- *CLIENT_OUTPUTONLY_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 Input-DeactivationStatus. 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`.

- *`CLIENT_SECURITY_HASH`*

  Contains client security hash (optional feature). This hash is used to check if this client is allowed to enter specified channels with a matching *`CHANNEL_SECURITY_SALT`*. Motivation is to enforce clients joining a server with the specific identity, nickname and metadata.

  Please see the chapter “Security salts and hashes” in the Server SDK documentation for details.

- *`CLIENT_ENCRYPTION_CIPHERS`*

  Comma-separated list of ciphers offered to the server to pick one from.

  Possible values are:

  ```
  "AES-128"
  "AES-256"
  ```

  Defaults to "AES-256,AES-128".

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);

uint64 serverConnectionHandlerID;
ClientProperties flag;
int value;
```

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

uint64 serverConnectionHandlerID;
ClientProperties flag;
const char* value;
```

- *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,
returnCode);

uint64 serverConnectionHandlerID;
const char* returnCode;
```

The idea behind flushing is, one can modify multiple values by calling `ts3client_setClientVariableAsString` and `ts3client_setClientVariableAsInt` and then apply all changes in one step.

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, NULL) != 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, NULL) != 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, NULL) != 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`*`);`

`uint64` *`serverConnectionHandlerID`*`;`

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

```
unsigned int ts3client_getClientVariableAsUInt64(serverConnectionHandlerID, clien-
tID, flag, result);

uint64 serverConnectionHandlerID;
anyID clientID;
ClientProperties flag;
uint64* result;
```

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

uint64 serverConnectionHandlerID;
anyID clientID;
ClientProperties flag;
char** result;
```

- *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, 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.

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);

uint64 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.

- *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. This event is also called everytime a client variable has been changed:

```
void onUpdateClientEvent(serverConnectionHandlerID, clientID, invokerID, invoker-
Name, invokerUniqueIdentifier);

uint64 serverConnectionHandlerID;
anyID clientID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

- *serverConnectionHandlerID*

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

- *clientID*

  ID of the client whose variables are now available or have changed.

- *invokerID*

  ID of the client who edited this clients variables.

- *invokerName*

  Nickname of the client who edited this clients variables.

- *invokerUniqueIdentifier*

  Unique ID of the client who edited this clients variables.

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 standard 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);
```

```
uint64 serverConnectionHandlerID;
anyID clientID;
const uint64* targetChannelIDArray;
const anyID* targetClientIDArray;
const char* returnCode;
```

- *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*

  Array of channel IDs, terminated with 0. These channels will be added to the whisper list.

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

- *targetClientIDArray*

  Array of client IDs, terminated with 0. 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).

To control which client is allowed to whisper to own client, the Client Lib implements an internal whisper whitelist mechanism. When a client recieves a whisper while the whispering client has not yet been added to the whisper allow list, the receiving client gets the following event. Note that whisper voice data is not received until the sending client is added to the receivers whisper allow list.

```
void onIgnoredWhisperEvent(serverConnectionHandlerID, clientID);

uint64 serverConnectionHandlerID;
anyID clientID;
```

- *serverConnectionHandlerID*

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

- *clientID*

  ID of the whispering client.

The receiving client can decide to allow whispering from the sender and add the sending client to the whisper allow list by calling `ts3client_allowWhispersFrom`. If the sender is not added by the receiving client, this event persists being called but no voice data is transmitted to the receiving client.

To add a client to the whisper allow list:

```
unsigned int ts3client_allowWhispersFrom(serverConnectionHandlerID, clID);

uint64 serverConnectionHandlerID;
anyID clID;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which the client should be added to the whisper allow list.

- *clID*

  ID of the client to be added to the whisper allow list.

To remove a client from the whisper allow list:

```
unsigned int ts3client_removeFromAllowedWhispersFrom(serverConnectionHandlerID,
clID);

uint64 serverConnectionHandlerID;
anyID clID;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which the client should be removed from the whisper allow list.

- *clID*

  ID of the client to be removed from the whisper allow list.

It won't have bad sideeffects if the same client ID is added to the whisper allow list multiple times.

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);

uint64 serverConnectionHandlerID;
uint64 channelID;
ChannelProperties flag;
int* result;
```

```
unsigned int ts3client_getChannelVariableAsUInt64(serverConnectionHandlerID, chan-
nelID, flag, result);

uint64 serverConnectionHandlerID;
uint64 channelID;
ChannelProperties flag;
uint64* result;
```

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

uint64 serverConnectionHandlerID;
uint64 channelID;
ChannelProperties flag;
char* result;
```

- *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 of type 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.

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_CODEC_LATENCY_FACTOR, //Available for all channels that are "in view", always up-to-date
  CHANNEL_CODEC_IS_UNENCRYPTED, //Available for all channels that are "in view", always up-to-date
  CHANNEL_SECURITY_SALT,        //Sets the options+salt for security hash (SDK only)
  CHANNEL_DELETE_DELAY,         //How many seconds to wait before deleting this channel
  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);

  uint64 serverConnectionHandlerID;
  uint64 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: Codec used for this channel:

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

- 3 - Celt (Mono, 48kHz)
- 4 - Opus Voice (Mono, 48khz)
- 5 - Opus Music (Stereo, 48khz)

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 / Permanent

    Semi-permanent and permanent channels are not automatically deleted when the last user left. As SDK servers are not persistant over restart, there is no effective difference between these two in the SDK.

- *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*.

- *CHANNEL_CODEC_LATENCY_FACTOR*

(Int: 1-10): Latency of this channel. This allows to increase the packet size resulting in less bandwidth usage at the cost of higher latency. A value of 1 (default) is the best setting for lowest latency and best quality. If bandwidth or network quality are restricted, increasing the latency factor can help stabilize the connection. Higher latency values are only possible for low-quality codec and codec quality settings.

For best voice quality a low latency factor is recommended.

- *CHANNEL_CODEC_IS_UNENCRYPTED*

  Int (0/1): If 1, this channel is not using encrypted voice data. If 0, voice data is encrypted for this channel. Note that channel voice data encryption can be globally disabled or enabled for the virtual server. Changing this flag makes only sense if global voice data encryption is set to be configured per channel as *CODEC_ENCRYPTION_PER_CHANNEL* (the default behaviour).

- *CHANNEL_SECURITY_SALT*

  Contains the channels security salt (optional feature). When a client connects, the clients hash value in *CLIENT_SECURITY_HASH* is check against the channel salt to allow or deny the client to join this channel. Motivation is to enforce clients joining a server with the specific identity, nickname and metadata.

  Please see the chapter “Security salts and hashes” in the Server SDK documentation for details.

- *CHANNEL_DELETE_DELAY*

  This parameter defines how many seconds the server waits until a temporary channel is deleted when empty.

  When a temporary channel is created, a timer is started. If a user joins the channel before the countdown is finished, the channel is not deleted. After the last person has left the channel, the countdown starts again. *CHANNEL_DELETE_DELAY* defines the length of this countdown in seconds.

  The time since the last client has left the temporary channel can be queried with `ts3client_getChannelEmptySecs` [94].

To modify channel data use

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

uint64 serverConnectionHandlerID;
uint64 channelID;
ChannelProperties flag;
int value;
```

```
unsigned int ts3client_setChannelVariableAsUInt64(serverConnectionHandlerID, chan-
nelID, flag, value);

uint64 serverConnectionHandlerID;
uint64 channelID;
ChannelProperties flag;
uint64 value;
```

```
unsigned int ts3client_setChannelVariableAsString(serverConnectionHandlerID, channelID, flag, value);

uint64 serverConnectionHandlerID;
uint64 channelID;
ChannelProperties flag;
const char* value;
```

- *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. Depending on which function is used, the value can be of type int, uint64 or string.

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);

uint64 serverConnectionHandlerID;
uint64 channelID;
```

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);

uint64 serverConnectionHandlerID;
uint64 channelID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

- *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);

uint64 serverConnectionHandlerID;
char** channelNameArray;
uint64* result;
```

- *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 voice data encryption

Voice data can be encrypted or unencrypted. Encryption will increase CPU load, so should be used only when required. Encryption can be configured per channel (the default) or globally enabled or disabled for the whole virtual server. By default channels are sending voice data unencrypted, newly created channels would need to be set to encrypted if required.

To configure the global virtual server encryption settings, modify the virtual server property *VIRTUALSERVER_CODEC_ENCRYPTION_MODE* to one of the following values:

```
enum CodecEncryptionMode {
   CODEC_ENCRYPTION_PER_CHANNEL = 0,  // Default
   CODEC_ENCRYPTION_FORCED_OFF,
   CODEC_ENCRYPTION_FORCED_ON,
};
```

Voice data encryption per channel can be configured by setting the channel property *CHANNEL_CODEC_IS_UNENCRYPTED* to 0 (encrypted) or 1 (unencrypted) if global encryption mode is *CODEC_ENCRYPTION_PER_CHANNEL*. If encryption is forced on or off globally, the channel property will be automatically set by the server.

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_getChannelVariableAsUInt64 or changed with ts3client_setChannelVariableAsUInt64.

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_setChannelVariableAsUInt64.

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 reading client and channel data, server information can be queried with

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

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

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

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

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

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

- *`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_CODEC_ENCRYPTION_MODE, //available and always up-to-date when connected
  VIRTUALSERVER_ENCRYPTION_CIPHERS,    //SDK only: list of ciphers that can be used for encryption
  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`.

- *VIRTUALSERVER_CODEC_ENCRYPTION_MODE*

  Defines if voice data encryption is configured per channel, globally forced on or globally forced off for this virtual server. The default behaviour is configure per channel, in this case modifying the channel property *CHANNEL_CODEC_IS_UNENCRYPTED* defines voice data encryption of individual channels.

  Virtual server encryption mode can be set to the following parameters:

  ```
  enum CodecEncryptionMode {
     CODEC_ENCRYPTION_PER_CHANNEL = 0,
     CODEC_ENCRYPTION_FORCED_OFF,
     CODEC_ENCRYPTION_FORCED_ON,
  };
  ```

  This property is always available when connected.

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`*`);`

`uint64` *`serverConnectionHandlerID`*`;`

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

```
unsigned int onServerUpdatedEvent(serverConnectionHandlerID);

uint64 serverConnectionHandlerID;
```

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

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

uint64 serverConnectionHandlerID;
anyID editerID;
const char* editerName;
const char* editerUniqueIdentifier;
```

- *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, clientID,
newChannelID, password, returnCode);
```

```
uint64 serverConnectionHandlerID;
anyID clientID;
uint64 newChannelID;
const char* password;
const char* returnCode;
```

- *serverConnectionHandlerID*

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

- *clientID*

  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. Pass an empty string if no password is given.

- *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, newChannelID, visibility, moveMessage);

uint64 serverConnectionHandlerID;
anyID clientID;
uint64 oldChannelID;
uint64 newChannelID;
int visibility;
const char* moveMessage;
```

- *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*

  When a client disconnects from the server, this includes the optional message set by the disconnecting client in `ts3client_stopConnection`.

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

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

Now wait for the callback:

```
void my_onClientMoveEvent(uint64 scHandlerID, anyID clientID,
                          uint64 oldChannelID, uint64 newChannelID,
                          int visibility, const char* moveMessage) {
  // 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
  // moveMessage   ->  Optional message set by disconnecting clients
}
```

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`*`);`

`uint64` *`serverConnectionHandlerID`*`;`

```
anyID clientID;
uint64 oldChannelID;
uint64 newChannelID;
int visibility;
anyID moverID;
const char* moverName;
const char* moverUniqueIdentifier;
const char* 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 client who initiated the move. *`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);
```

```
uint64 serverConnectionHandlerID;
uint64 channelParentID;
```

- *`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);
```

```
uint64 serverConnectionHandlerID;
uint64 channelID;
uint64 channelParentID;
```

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

- *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(uint64 scHandlerID, uint64 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_setChannelVariableAsUInt64(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);

uint64 serverConnectionHandlerID;
uint64 channelID;
int force;
const char* returnCode;
```

- *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 delete 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);

uint64 serverConnectionHandlerID;
uint64 channelID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

- *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);

uint64 serverConnectionHandlerID;
uint64 channelID;
uint64 newChannelParentID;
uint64 newChannelOrder;
const char* returnCode;
```

- *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);

uint64 serverConnectionHandlerID;
uint64 channelID;
uint64 newChannelParentID;
anyID invokerID;
const char* invokerName;
const char* invokerUniqueIdentifier;
```

- *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 a client, channel or virtual server. Depending on the target, there are three functions to send text messages and one callback to receive them.

Sending

To send a private text message to a client:

```
unsigned int ts3client_requestSendPrivateTextMsg(serverConnectionHandlerID, message,
targetClientID, returnCode);

uint64 serverConnectionHandlerID;
```

```
const char* message;
anyID targetClientID;
const char* returnCode;
```

- *serverConnectionHandlerID*

  Id of the target server connection handler.

- *message*

  String containing the text message

- *targetClientID*

  Id of the target client.

- *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 send a text message to a channel:

```
unsigned int ts3client_requestSendChannelTextMsg(serverConnectionHandlerID, message,
targetChannelID, returnCode);
```

```
uint64 serverConnectionHandlerID;
const char* message;
anyID targetChannelID;
const char* returnCode;
```

- *serverConnectionHandlerID*

  Id of the target server connection handler.

- *message*

  String containing the text message

- *targetChannelID*

  Id of the target channel.

- *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 send a text message to the virtual server:

```
unsigned int ts3client_requestSendServerTextMsg(serverConnectionHandlerID, message, returnCode);

uint64 serverConnectionHandlerID;
const char* message;
const char* returnCode;
```

- *serverConnectionHandlerID*

  Id of the target server connection handler.

- *message*

  String containing the text message

- *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 client with ID 123:

```
const char *msg = "Hello TeamSpeak!";
anyID targetClientID = 123;

if(ts3client_requestSendPrivateTextMsg(scHandlerID, msg, targetClient, NULL) != ERROR_ok) {
  /* Handle error */
}
```

Receiving

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

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

uint64 serverConnectionHandlerID;
anyID targetMode;
anyID toID;
anyID fromID;
const char* fromName;
const char* fromUniqueIdentifier;
const char* message;
```

- *serverConnectionHandlerID*

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

- *targetMode*

  Target mode of this text message. The value is defined by the enum TextMessageTargetMode:

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

- *toID*

  Id of the target of the text message.

- *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.

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, clientID, kickReason, returnCode);

uint64 serverConnectionHandlerID;
anyID clientID;
const char* kickReason;
const char* returnCode;
```

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

uint64 serverConnectionHandlerID;
anyID clientID;
const char* kickReason;
const char* returnCode;
```

- *serverConnectionHandlerID*

Id of the target server connection.

- *clientID*

  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);
```

```
uint64 serverConnectionHandlerID;
anyID clientID;
uint64 oldChannelID;
uint64 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);
```

```
uint64 serverConnectionHandlerID;
anyID clientID;
uint64 oldChannelID;
uint64 newChannelID;
int visibility;
anyID kickerID;
const char* kickerName;
const char* kickerUniqueIdentifier;
const char* kickMessage;
```

- *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. It would also be possible to always subscribe to all channels to get notifications about all clients on the server.

Subscriptions are meant to have a flexible way to balance bandwidth usage. On a crowded server limiting the number of subscribed channels is a way to reduce network traffic. Also subscriptions allow to usage “private” channels, whose members cannot be seen by other users.

Note

A client is automatically subscribed to the current channel.

To subscribe to a list of channels (zero-terminated array of channel IDs) call:

```
unsigned int ts3client_requestChannelSubscribe(serverConnectionHandlerID, channelI-
DArray, returnCode);

uint64 serverConnectionHandlerID;
```

```
const uint64* channelIDArray;
const char* returnCode;
```

To unsubscribe from a list of channels (zero-terminated array of channel IDs) call:

```
unsigned int ts3client_requestChannelUnsubscribe(serverConnectionHandlerID, channel-
IDArray, returnCode);

uint64 serverConnectionHandlerID;
const uint64* channelIDArray;
const char* returnCode;
```

To subscribe to all channels on the server call:

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

uint64 serverConnectionHandlerID;
const char* returnCode;
```

To unsubscribe from all channels on the server call:

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

uint64 serverConnectionHandlerID;
const char* returnCode;
```

To check if a channel is currently subscribed, check the channel property *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);

uint64 serverConnectionHandlerID;
uint64 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);

uint64 serverConnectionHandlerID;
```

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

```
void onChannelUnsubscribeEvent(serverConnectionHandlerID, channelID);

uint64 serverConnectionHandlerID;
uint64 channelID;
```

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

```
void onChannelUnsubscribeFinishedEvent(serverConnectionHandlerID);

uint64 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);

uint64 serverConnectionHandlerID;
anyID clientID;
uint64 oldChannelID;
uint64 newChannelID;
int visibility;
```

- *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);

uint64 serverConnectionHandlerID;
const anyID* clientIDArray;
const char* returnCode;
```

To unmute one or more clients:

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

uint64 serverConnectionHandlerID;
const anyID* clientIDArray;
const char* returnCode;
```

- *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;
```

- *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;
```

- *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;
    }
}
```

Custom passwords

The TeamSpeak SDK has the optional ability to do custom password handling. This makes it possible to allow people on the server (or channels) with passwords that are checked against outside datasources, like LDAP or other databases.

To implement custom password, both server and client need to add custom callbacks, which will be spontaneously called whenever a password check is done in TeamSpeak. The SDK developer can implement own checks to validate the password instead of using the TeamSpeak built-in mechanism.

Both Server and Client Lib can implement the following callback to encrypt a user password. This function is called in the Client Lib when a channel password is set.

This can be used to hash the password in the same way it is hashed in the outside data store. Or just copy the password to send the clear text to the server.

```
void onClientPasswordEncrypt(serverID, plaintext, encryptedText, encryptedTextByte-
Size);

uint64 serverID;
const char* plaintext;
char* encryptedText;
int encryptedTextByteSize;
```

- *serverID*

  ID of the server the password call occured

- *plaintext*

  The plaintext password

- *encryptedText*

  Fill with your custom encrypted password. Must be a 0-terminated string with a size not larger than *encrypted-TextByteSize*.

- *encryptedTextByteSize*

  Size of the buffer pointed to by *encryptedText*.

Other events

When a client starts or stops talking, a talk status change event is sent by the server:

```
void onTalkStatusChangeEvent(serverConnectionHandlerID, status, isReceivedWhisper,
clientID);

uint64 serverConnectionHandlerID;
int status;
int isReceivedWhisper;
anyID clientID;
```

- *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_NOT_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 or just ignore this value.

- *isReceivedWhisper*

  1 if the talk event was caused by whispering, 0 if caused by normal talking.

- *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);

uint64 serverConnectionHandlerID;
anyID clientID;
uint64 oldChannelID;
uint64 newChannelID;
int visibility;
const char* timeoutMessage;
```

- *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);

uint64 serverConnectionHandlerID;
uint64 channelID;
```

- *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)`.

The following 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);

uint64 serverConnectionHandlerID;
uint64 channelID;
```

- *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);

uint64 serverConnectionHandlerID;
uint64 channelID;
```

- *serverConnectionHandlerID*

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

- *channelID*

  ID of the channel with the changed password.

Miscellaneous functions

Memory dynamically allocated in the Client Lib needs to be released with:

```
unsigned int ts3client_freeMemory(pointer);

void* pointer;
```

- *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);

uint64 serverConnectionHandlerID;
intstatus;
```

- *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`.

With the delayed temporary channel deletion feature, users can define after how many seconds a temporary channel will be deleted after the last client has left the channel. The delay is defined by setting the channel variable *CHANNEL_DELETE_DELAY*. This variable can be set and queried as described in channel information.

To query the time in seconds since the last client has left a temporary channel, call:

```
unsigned int ts3client_getChannelEmptySecs(serverConnectionHandlerID, channelID, result);

uint64 serverConnectionHandlerID;
uint64 channelID;
int* result;
```

- *serverConnectionHandlerID*

  ID of the server connection handler on which the time should be queried.

- *channelID*

  ID of the channel to query.

- *result*

  Address of a variable that receives the time in seconds.

Filetransfer

The TeamSpeak SDK includes the ability to support filetransfer, like the regular TeamSpeak server and client offer. The Server can function as a file storage, which can be accessed by Clients who can up- and download files. Files are stored on the filesystem where the server is running.

In general, clients can initiate filetransfer actions like uploading or downloading a file, requesting file information (size, name, path etc.), list files in a directory and so on. The functions to call these actions are explained in detail below. In addition to the functions actively called, there are filetransfer related callbacks which are triggered when the server returned the requested information (e.g. list of files in a directory).

Each transfer is identified by a *transferID*, which is passed to most filetransfer functions. Transfer IDs are unique during the time of the transfer, but may be reused again some time after the previous transfer with the same ID has finished.

Files are organized on the server inside channels (identified by their *channelID*. The top-level directory in each channel is "/". Subdirectories in each channel may exist and are defined with a path of the form "/dir1/dir2". Subdirectories are optional and need to be created with `ts3client_requestCreateDirectory`, the channel root directory always exists by default.

Query information

The following functions allow to query information about a file transfer identified by its *transferID*.

Query the file name of the specified transfer:

```
unsigned int ts3client_getTransferFileName(transferID, result);

anyID transferID;
char** result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Points to a C string containing the file name. Remember to call `ts3client_freeMemory` to release the string, which is dynamically allocated in the clientlib.

Query the file path of the specified transfer:

```
unsigned int ts3client_getTransferFilePath(transferID, result);

anyID transferID;
char** result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Points to a C string containing the file path. Remember to call `ts3client_freeMemory` to release the string, which is dynamically allocated in the clientlib.

Query the remote path on the server of the specified transfer:

```
unsigned int ts3client_getTransferFileRemotePath(transferID, result);

anyID transferID;
char** result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Points to a C string containing the remote path on the server. Remember to call `ts3client_freeMemory` to release the string, which is dynamically allocated in the clientlib.

Query the file size of the specified transfer:

```
unsigned int ts3client_getTransferFileSize(transferID, result);

anyID transferID;
uint64* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  File size of the transfer.

Query the currently transferred file size of the queried transfer:

```
unsigned int ts3client_getTransferFileSizeDone(transferID, result);

anyID transferID;
uint64* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Already transferred size of the transfer.

Query if the specified transfer is an upload or download:

```
unsigned int ts3client_isTransferSender(transferID, result);

anyID transferID;
int* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  1 == upload, 0 == download

Query the status of the specified transfer:

```
unsigned int ts3client_getTransferStatus(transferID, result);

anyID transferID;
int* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Current status of the file transfer, specified by the struct *FileTransferState*:

```
enum FileTransferState {
    FILETRANSFER_INITIALISING = 0,
    FILETRANSFER_ACTIVE,
    FILETRANSFER_FINISHED,
};
```

Query the current speed of the specified transfer:

```
unsigned int ts3client_getCurrentTransferSpeed(transferID, result);

anyID transferID;
float* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Currently measured speed of the file transfer.

Query the average speed of the specified transfer:

```
unsigned int ts3client_getAverageTransferSpeed(transferID, result);

anyID transferID;
float* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

Average speed of the file transfer.

Query the time the specified transfer has used:

```
unsigned int ts3client_getTransferRunTime(transferID, result);

anyID transferID;
uint64* result;
```

- *transferID*

  ID of the filetransfer we want to query.

- *result*

  Time the transfer has used.

Initiate transfers

The following functions implement the core functionality of filetransfers. They initiate new up- and downloads, request file info, delete and rename files, create directories, list directories etc.

Request uploading a local file to the server:

```
unsigned int ts3client_sendFile(serverConnectionHandlerID, channelID, channelPW,
file, overwrite, resume, sourceDirectory, result, returnCode);

uint64 serverConnectionHandlerID;
uint64 channelID;
const char* channelPW;
const char* file;
int overwrite;
int resume;
const char* sourceDirectory;
anyID* result;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *channelID*

  Target channel ID in which the file should be uploaded.

- *channelPW*

Optional channel password. Pass empty string if unused.

- *file*

  Filename of the local file, which is to be uploaded.

- *overwrite*

  1 == overwrite remote file if it exists, 0 = do not overwrite (operation will abort if remote file exists)

- *resume*

  If we have a previously halted transfer: 1 = resume, 0 = restart transfer

- *sourceDirectory*

  Local directory where the file to upload is located.

- *result*

  Pointer to memory where the transferID will be stored, if the transfer has been started successfully (when this function returns *ERROR_ok*).

- *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.

Request downloading a file from the server:

```
unsigned int ts3client_requestFile(serverConnectionHandlerID, channelID, channelPW,
file, overwrite, resume, destinationDirectory, result, returnCode);
```

```
uint64 serverConnectionHandlerID;
uint64 channelID;
const char* channelPW;
const char* file;
int overwrite;
int resume;
const char* destinationDirectory;
anyID* result;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *channelID*

  Remote channel ID from which the file should be downloaded.

- *channelPW*

  Optional channel password. Pass empty string if unused.

- *file*

  Filename of the remote file, which is to be downloaded.

- *overwrite*

  1 == overwrite local file if it exists, 0 = do not overwrite (operation will abort if local file exists)

- *resume*

  If we have a previously halted transfer: 1 = resume, 0 = restart transfer

- *destinationDirectory*

  Local target directory name where the download file should be saved.

- *result*

  Pointer to memory where the transferID will be stored, if the transfer has been started successfully (when this function returns *ERROR_ok*).

- *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.

Pause a transfer, specified by its *transferID*:

```
unsigned int ts3client_haltTransfer(serverConnectionHandlerID, transferID, delete-
UnfinishedFile, returnCode);

uint64 serverConnectionHandlerID;
anyID transferID;
int deleteUnfinishedFile;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *transferID*

  ID of the transfer that should be halted.

- *deleteUnfinishedFile*

  1 = delete the halted file, 0 = do not deleted halted file

- *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.

Query list of files in a directory. The answer from the server will trigger the `onFileListEvent` and `onFileListFinishedEvent` callbacks with the requested information.

```
unsigned int ts3client_requestFileList(serverConnectionHandlerID, channelID, channelPW, path, returnCode);

uint64 serverConnectionHandlerID;
uint64 channelID;
const char* channelPW;
const char* path;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *channelID*

  Remote channel ID, from which we want to query the file list.

- *channelPW*

  Optional channel password. Pass empty string if unused.

- *path*

  Path inside the channel, defining the subdirectory. Top level path is "/"

- *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.

Query information of a specified file. The answer from the server will trigger the `onFileInfoEvent` callback with the requested information.

```
unsigned int ts3client_requestFileInfo(serverConnectionHandlerID, channelID, channelPW, file, returnCode);

uint64 serverConnectionHandlerID;
uint64 channelID;
const char* channelPW;
```

```
const char* file;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *channelID*

  Remote channel ID, from which we want to query the file info.

- *channelPW*

  Optional channel password. Pass empty string if unused.

- *file*

  File name we want to request info from, needs to include the full path within the channel, e.g. “/file” for a top-level file or “/dir1/dir2/file” for a file located in a subdirectory.

- *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.

Request deleting one or more remote files on the server:

```
unsigned int ts3client_requestDeleteFile(serverConnectionHandlerID, channelID, channelPW, file, returnCode);

uint64 serverConnectionHandlerID;
uint64 channelID;
const char* channelPW;
const char** file;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *channelID*

  Remote channel ID, in which we want to delete the files.

- *channelPW*

  Optional channel password. Pass empty string if unused.

- *file*

List of files we request to delete. Array must be NULL-terminated. The file names need to include the full path within the channel, e.g. "/file" for a top-level file or "/dir1/dir2/file" for a file located in a subdirectory.

- *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.

Request creating a directory:

```
unsigned int ts3client_requestCreateDirectory(serverConnectionHandlerID, channelID,
channelPW, directoryPath, returnCode);
```

```
uint64 serverConnectionHandlerID;
uint64 channelID;
const char* channelPW;
const char* directoryPath;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *channelID*

  Remote channel ID, in which we want to create the directory.

- *channelPW*

  Optional channel password. Pass empty string if unused.

- *file*

  Name of the directory to create. The directory name needs to include the full path within the channel, e.g. "/file" for a top-level file or "/dir1/dir2/file" for a file located in a subdirectory.

- *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.

Request renaming or moving a file. If the source and target channels and paths are the same, the file will simply be renamed.

```
unsigned int ts3client_requestRenameFile(serverConnectionHandlerID, fromChannelID,
fromChannelPW, toChannelID, toChannelPW, oldFile, newFile, returnCode);
```

```
uint64 serverConnectionHandlerID;
```

```
uint64 fromChannelID;
const char* fromChannelPW;
uint64 toChannelID;
const char* toChannelPW;
const char* oldFile;
const char* newFile;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server the file transfer operation will be requested.

- *fromChannelID*

  Source channel ID, in which we want to rename the file.

- *fromChannelPW*

  Optional source channel password. Pass empty string if unused.

- *toChannelID*

  Target channel ID, to which we want to move the file. If the file should not be moved to another channel, this parameter should be equal to *fromChannelID*.

- *toChannelPW*

  Optional target channel password. Pass empty string if unused.

- *oldFile*

  Old name of the file. The file name needs to include the full path within the channel, e.g. “/file” for a top-level file or “/dir1/dir2/file” for a file located in a subdirectory.

- *newFile*

  Target name of the directory to create. The directory name need to include the full path within the channel, e.g. “/file” for a top-level file or “/dir1/dir2/file” for a file located in a subdirectory.

  To move files to another subdirectory in the same channel without renaming the file, *fromChannelID* has to be equal to *toChannelID*, keep the file name itself but just change the path.

- *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.

Speed limits

The TeamSpeak SDK offers the possibility to control and finetune transfer speed limits. These limits can be applied to the complete server, specific virtual servers or for each individual transfer. By default the transfer speed is unlimited. Every file transfer should at least have a minimum speed limit of 5kb/s.

Neither the TeamSpeak client nor server will store any of those values. When used, they'll have to be set at each client start to be considered permanent.

To set the upload speed limit for all virtual servers in bytes/s:

```
unsigned int ts3client_setInstanceSpeedLimitUp(newLimit);

uint64 newLimit;
```

To set the download speed limit for all virtual servers in bytes/s:

```
unsigned int ts3client_setInstanceSpeedLimitDown(newLimit);

uint64 newLimit;
```

To get the upload speed limit for all virtual servers in bytes/s:

```
unsigned int ts3client_getInstanceSpeedLimitUp(limit);

uint64* limit;
```

To get the download speed limit for all virtual servers in bytes/s:

```
unsigned int ts3client_getInstanceSpeedLimitDown(limit);

uint64* limit;
```

To set the upload speed limit for the specified virtual server in bytes/s:

```
unsigned                                                                    int
ts3client_setServerConnectionHandlerSpeedLimitUp(serverConnectionHandlerID, newLim-
it);

uint64 serverConnectionHandlerID;
uint64 newLimit;
```

To set the download speed limit for the specified virtual server in bytes/s:

```
unsigned                                                                    int
ts3client_setServerConnectionHandlerSpeedLimitDown(serverConnectionHandlerID,
newLimit);

uint64 serverConnectionHandlerID;
uint64 newLimit;
```

To get the upload speed limit for the specified virtual server in bytes/s:

```
unsigned int
ts3client_getServerConnectionHandlerSpeedLimitUp(serverConnectionHandlerID, limit);

uint64 serverConnectionHandlerID;
uint64* limit;
```

To get the download speed limit for the specified virtual server in bytes/s:

```
unsigned int
ts3client_getServerConnectionHandlerSpeedLimitDown(serverConnectionHandlerID, limit);

uint64 serverConnectionHandlerID;
uint64* limit;
```

To set the up- or download speed limit for the specified file transfer in bytes/s. Use `ts3client_isTransferSender` to query if the transfer is an up- or download.

```
unsigned int ts3client_setTransferSpeedLimit(transferID, newLimit);

anyID transferID;
uint64 newLimit;
```

To get the speed limit for the specified file transfer in bytes/s:

```
unsigned int ts3client_getTransferSpeedLimit(transferID, limit);

anyID transferID;
uint64* limit;
```

Callbacks

This event is called when a file transfer, triggered by `ts3client_sendFile` or `ts3client_requestFile` has finished or aborted with an error.

```
void onFileTransferStatusEvent(transferID, status, statusMessage, remotefileSize,
serverConnectionHandlerID);

anyID transferID;
unsigned int status;
const char* statusMessage;
uint64 remotefileSize;
```

```
uint64 serverConnectionHandlerID;
```

- *transferID*

  ID of the transfer. This ID was returned by the call to `ts3client_sendFile` or `ts3client_requestFile` which triggered this event.

- *status*

  Indicates how and why the transfer has finished:

  - *ERROR_file_transfer_complete*

    Transfer completed successfully.

  - *ERROR_file_transfer_canceled*

    Transfer was halted by a call to `ts3client_haltTransfer`.

  - *ERROR_file_transfer_interrupted*

    An error occured, transfer was stopped for various reasons (network error etc.)

  - *ERROR_file_transfer_reset*

    Transfer was reset. This can happen if the remote file has changed (another user uploaded another file under the same channel ID, path and file name).

- *statusMessage*

  Status text message for a verbose display of the *status* parameter.

- *remotefileSize*

  Remote size of the file on the server.

- *serverConnectionHandlerID*

  ID of the virtual server on which the file list was requested.

Callback containing the reply by the server on ts3client_requestFileList. There event is called for every file in the specified path. After the last file, `onFileListFinished` will indicate the end of the list.

```
void onFileListEvent(serverConnectionHandlerID, channelID, path, name, size, date-
time, type, incompletesize, returnCode);

uint64 serverConnectionHandlerID;
uint64 channelID;
const char* path;
const char* name;
uint64 size;
uint64 datetime;
```

```
int type;
uint64 incompletesize;
const char* returnCode;
```

- *serverConnectionHandlerID*

  ID of the virtual server on which the file list was requested.

- *channelID*

  ID of the channel which file list was requested.

- *path*

  Subdirectory inside the channel for which the file list was requested. "/" indicates the root directory is listed.

- *name*

  File name.

- *size*

  File size

- *datetime*

  File date (Unix time in seconds)

- *type*

  Indicates if this entry is a directory or a file. Type is specified as:

  ```
  enum {
      FileListType_Directory = 0,
      FileListType_File,
  };
  ```

- *incompletesize*

  If the file is currently still being transferred, this indicates the currently transferred file size.

- *returnCode*

  String containing the return code if it has been set by `ts3client_requestFileList` which triggered this event.

Callback indicating the end of an incoming file list, see `onFileList`.

```
void onFileListFinishedEvent(serverConnectionHandlerID, channelID, path);
```

```
uint64 serverConnectionHandlerID;
uint64 channelID;
const char* path;
```

- *serverConnectionHandlerID*

  ID of the virtual server on which the file list was requested.

- *channelID*

  If of the channel which files have been listed.

- *path*

  Path within the channel which files have been listed.

Callback containing the reply by the server for ts3client_requestFileInfo:

```
void onFileInfoEvent(serverConnectionHandlerID, channelID, name, size, datetime);

uint64 serverConnectionHandlerID;
uint64 channelID;
const char* name;
uint64 size;
uint64 datetime;
```

- *serverConnectionHandlerID*

  ID of the virtual server on which the file info was requested.

- *channelID*

  If of the channel in which the file is located.

- *name*

  File name including the path within the channel in which the file is located.

- *size*

  File size

- *datetime*

  File date (Unix time in seconds)

FAQ

- How to implement Push-To-Talk?
- How to adjust the volume?
- How to talk across channels?

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, NULL) != 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");
```

How to adjust the volume?

Output volume

The global 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 increase the output volume by 10 decibel:

```
ts3client_setPlaybackConfigValue(scHandlerID, "volume_modifier", 10);
```

In addition to modifying the global output volue, the volume of individual clients can be changed with `ts3client_setClientVolumeModifier`.

Input volume

Automatic Gain Control (AGC) takes care of the input volume during preprocessing automatically. Instead of modifying the input volume directly, you modify the AGC preprocessor settings with `setProProcessorConfigValue`.

How to talk across channels?

Generally clients can only talk to other clients in the same channel. However, for specific scenarios this can be overruled using whisper lists.. This feature allows specific clients to temporarily talk to other clients or channels outside of their own channel. While whispering, talking to the own channel is disabled.

An example for a scenario where whisper may be useful would be a team consisting of a number of squads. Each squad is assigned to one channel, so squad members can only talk to other members of the same squad. In addition, there is a team leader and squad leaders, who want to communicate accross the squad channels. This can be implemented with whispering, so the team leader could broadcast to all squad leaders, or a squad leader could briefly report to the team leader temporarily sending his voice data to him instead of the squad leaders channel.

This mechanism is powerful and flexible allowing the SDK developer to handle more complex scenarios overruling the standard behaviour where clients can only talk to other clients within the same channel.

Index

Symbols

A

B

C

E

F

H

L

M

N

P

R

S

U

V

W