libwsv5

Generated by Doxygen 1.9.4

Chapter 1

Data Structure Index

1.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

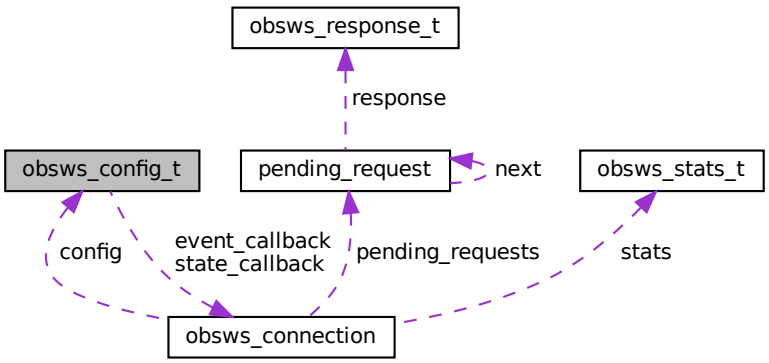
Data Structure Documentation

3.1 obsws_config_t Struct Reference

Connection configuration structure.

`#include <library.h>`

Collaboration diagram for obsws_config_t:



Data Fields

- const char * host
- uint16_t port
- const char * password
- bool use_ssl
- uint32_t connect_timeout_ms
- uint32_t recv_timeout_ms
- uint32_t send_timeout_ms
- uint32_t ping_interval_ms
- uint32_t ping_timeout_ms

- bool auto_reconnect
- uint32_t reconnect_delay_ms
- uint32_t max_reconnect_delay_ms
- uint32_t max_reconnect_attempts
- obsws_log_callback_t log_callback
- obsws_event_callback_t event_callback
- obsws_state_callback_t state_callback
- void * user_data

3.1.1 Detailed Description

Connection configuration structure.

This structure holds all the settings for connecting to OBS. You should fill this out with your specific needs, then pass it to obsws_connect(). A good starting point is to call obsws_config_init() which fills it with reasonable defaults, then only change the fields you care about (usually just host, port, and password).

Design note: We use a config struct instead of many function parameters because it's more flexible - adding new configuration options doesn't break existing code. It also makes it clear what options are available.

Definition at line 237 of file library.h.

3.1.2 Field Documentation

3.1.2.1 auto_reconnect

```
bool obsws_config_t::auto_reconnect
```

Definition at line 261 of file library.h.

3.1.2.2 connect_timeout_ms

```
uint32_t obsws_config_t::connect_timeout_ms
```

Definition at line 247 of file library.h.

3.1.2.3 event_callback

```
obsws_event_callback_t obsws_config_t::event_callback
```

Definition at line 270 of file library.h.

3.1.2.4 host

```
const char* obsws_config_t::host
```

Definition at line 239 of file library.h.

3.1.2.5 log_callback

```
obsws_log_callback_t obsws_config_t::log_callback
```

Definition at line 269 of file library.h.

3.1.2.6 max_reconnect_attempts

```
uint32_t obsws_config_t::max_reconnect_attempts
```

Definition at line 264 of file library.h.

3.1.2.7 max_reconnect_delay_ms

```
uint32_t obsws_config_t::max_reconnect_delay_ms
```

Definition at line 263 of file library.h.

3.1.2.8 password

```
const char* obsws_config_t::password
```

Definition at line 241 of file library.h.

3.1.2.9 ping_interval_ms

```
uint32_t obsws_config_t::ping_interval_ms
```

Definition at line 254 of file library.h.

3.1.2.10 ping_timeout_ms

```
uint32_t obsws_config_t::ping_timeout_ms
```

Definition at line 255 of file library.h.

3.1.2.11 port

```
uint16_t obsws_config_t::port
```

Definition at line 240 of file library.h.

3.1.2.12 reconnect_delay_ms

```
uint32_t obsws_config_t::reconnect_delay_ms
```

Definition at line 262 of file library.h.

3.1.2.13 recv_timeout_ms

```
uint32_t obsws_config_t::recv_timeout_ms
```

Definition at line 248 of file library.h.

3.1.2.14 send_timeout_ms

```
uint32_t obsws_config_t::send_timeout_ms
```

Definition at line 249 of file library.h.

3.1.2.15 state_callback

```
obsws_state_callback_t obsws_config_t::state_callback
```

Definition at line 271 of file library.h.

3.1.2.16 use_ssl

```
bool obsws_config_t::use_ssl
```

Definition at line 242 of file library.h.

3.1.2.17 user_data

```
void* obsws_config_t::user_data
```

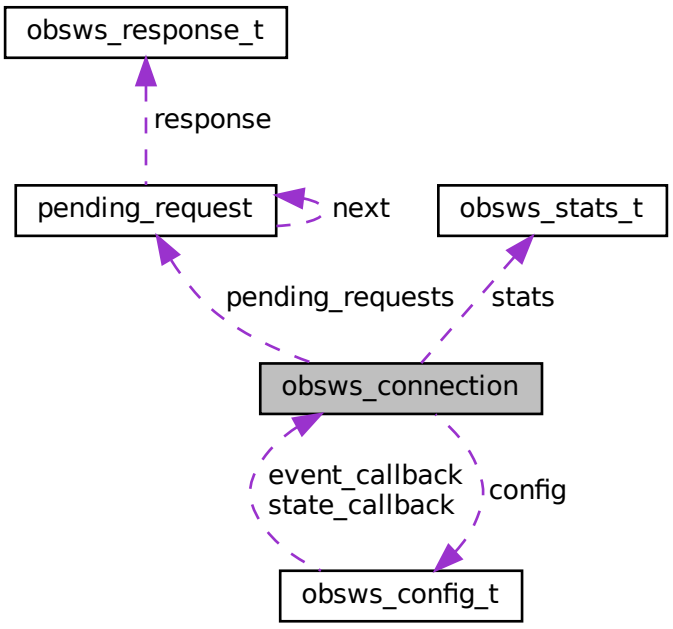
Definition at line 272 of file library.h.

The documentation for this struct was generated from the following file:

- library.h

3.2 obsws_connection Struct Reference

Collaboration diagram for obsws_connection:



Data Fields

- obsws_config_t config
- obsws_state_t state
- pthread_mutex_t state_mutex
- struct lws_context * lws_context
- struct lws * wsi
- char * recv_buffer
- size_t recv_buffer_size
- size_t recv_buffer_used
- char * send_buffer
- size_t send_buffer_size
- pthread_t event_thread
- pthread_mutex_t send_mutex
- bool thread_running
- bool should_exit
- pending_request_t * pending_requests
- pthread_mutex_t requests_mutex
- obsws_stats_t stats
- pthread_mutex_t stats_mutex
- time_t last_ping_sent
- time_t last_pong_received
- uint32_t reconnect_attempts
- uint32_t current_reconnect_delay
- bool auth_required
- char * challenge
- char * salt
- char * current_scene
- pthread_mutex_t scene_mutex

3.2.1 Detailed Description

Definition at line 169 of file library.c.

3.2.2 Field Documentation

3.2.2.1 auth_required

```
bool obsws_connection::auth_required
```

Definition at line 224 of file library.c.

3.2.2.2 challenge

```
char* obsws_connection::challenge
```

Definition at line 225 of file library.c.

3.2.2.3 config

```
obsws_config_t obsws_connection::config
```

Definition at line 171 of file library.c.

3.2.2.4 current_reconnect_delay

```
uint32_t obsws_connection::current_reconnect_delay
```

Definition at line 219 of file library.c.

3.2.2.5 current_scene

```
char* obsws_connection::current_scene
```

Definition at line 231 of file library.c.

3.2.2.6 event_thread

```
pthread_t obsws_connection::event_thread
```

Definition at line 194 of file library.c.

3.2.2.7 last_ping_sent

```
time_t obsws_connection::last_ping_sent
```

Definition at line 212 of file library.c.

3.2.2.8 last_pong_received

```
time_t obsws_connection::last_pong_received
```

Definition at line 213 of file library.c.

3.2.2.9 lws_context

```
struct lws_context* obsws_connection::lws_context
```

Definition at line 178 of file library.c.

3.2.2.10 pending_requests

```
pending_request_t* obsws_connection::pending_requests
```

Definition at line 202 of file library.c.

3.2.2.11 reconnect_attempts

```
uint32_t obsws_connection::reconnect_attempts
```

Definition at line 218 of file library.c.

3.2.2.12 recv_buffer

```
char* obsws_connection::recv_buffer
```

Definition at line 184 of file library.c.

3.2.2.13 recv_buffer_size

```
size_t obsws_connection::recv_buffer_size
```

Definition at line 185 of file library.c.

3.2.2.14 recv_buffer_used

```
size_t obsws_connection::recv_buffer_used
```

Definition at line 186 of file library.c.

3.2.2.15 requests_mutex

```
pthread_mutex_t obsws_connection::requests_mutex
```

Definition at line 203 of file library.c.

3.2.2.16 salt

```
char* obsws_connection::salt
```

Definition at line 226 of file library.c.

3.2.2.17 scene_mutex

```
pthread_mutex_t obsws_connection::scene_mutex
```

Definition at line 232 of file library.c.

3.2.2.18 send_buffer

```
char* obsws_connection::send_buffer
```

Definition at line 188 of file library.c.

3.2.2.19 send_buffer_size

```
size_t obsws_connection::send_buffer_size
```

Definition at line 189 of file library.c.

3.2.2.20 send_mutex

```
pthread_mutex_t obsws_connection::send_mutex
```

Definition at line 195 of file library.c.

3.2.2.21 should_exit

```
bool obsws_connection::should_exit
```

Definition at line 197 of file library.c.

3.2.2.22 state

```
obsws_state_t obsws_connection::state
```

Definition at line 174 of file library.c.

3.2.2.23 state_mutex

```
pthread_mutex_t obsws_connection::state_mutex
```

Definition at line 175 of file library.c.

3.2.2.24 stats

```
obsws_stats_t obsws_connection::stats
```

Definition at line 206 of file library.c.

3.2.2.25 stats_mutex

```
pthread_mutex_t obsws_connection::stats_mutex
```

Definition at line 207 of file library.c.

3.2.2.26 thread_running

```
bool obsws_connection::thread_running
```

Definition at line 196 of file library.c.

3.2.2.27 wsi

```
struct lws* obsws_connection::wsi
```

Definition at line 179 of file library.c.

The documentation for this struct was generated from the following file:

- library.c

3.3 obsws_response_t Struct Reference

Response structure for requests to OBS.

```
#include <library.h>
```

Data Fields

- bool success
- int status_code
- char * error_message
- char * response_data

3.3.1 Detailed Description

Response structure for requests to OBS.

When you send a request like obsws_set_current_scene(), you can get back a response with the result. The response tells you if it succeeded, and if not, why it failed. It might also contain response data from OBS - for example, obsws_get_current_scene() puts the scene name in response_data as JSON.

If you don't care about the response, you can pass NULL and not get one back. Otherwise you must free it with obsws_response_free() when done.

Design note: Responses are returned as strings instead of parsed JSON to save CPU - different callers care about different response fields, so we let them parse what they need. This also avoids dependency bloat.

Definition at line 309 of file library.h.

3.3.2 Field Documentation

3.3.2.1 error_message

```
char* obsws_response_t::error_message
```

Definition at line 312 of file library.h.

3.3.2.2 response_data

```
char* obsws_response_t::response_data
```

Definition at line 313 of file library.h.

3.3.2.3 status_code

```
int obsws_response_t::status_code
```

Definition at line 311 of file library.h.

3.3.2.4 success

```
bool obsws_response_t::success
```

Definition at line 310 of file library.h.

The documentation for this struct was generated from the following file:

- library.h

3.4 obsws_stats_t Struct Reference

Connection statistics - useful for monitoring and debugging connection health.

```
#include <library.h>
```

Data Fields

- uint64_t messages_sent
- uint64_t messages_received
- uint64_t bytes_sent
- uint64_t bytes_received
- uint64_t reconnect_count
- uint64_t error_count
- uint64_t last_ping_ms
- time_t connected_since

3.4.1 Detailed Description

Connection statistics - useful for monitoring and debugging connection health.

These stats let you see what's happening on the connection - how many messages have been sent/received, error counts, latency, etc. Useful for monitoring the connection quality, detecting if something is wrong, or just being curious about protocol activity. You get these by calling obsws_get_stats().

Definition at line 283 of file library.h.

3.4.2 Field Documentation

3.4.2.1 bytes_received

```
uint64_t obsws_stats_t::bytes_received
```

Definition at line 287 of file library.h.

3.4.2.2 bytes_sent

```
uint64_t obsws_stats_t::bytes_sent
```

Definition at line 286 of file library.h.

3.4.2.3 connected_since

```
time_t obsws_stats_t::connected_since
```

Definition at line 291 of file library.h.

3.4.2.4 error_count

```
uint64_t obsws_stats_t::error_count
```

Definition at line 289 of file library.h.

3.4.2.5 last_ping_ms

```
uint64_t obsws_stats_t::last_ping_ms
```

Definition at line 290 of file library.h.

3.4.2.6 messages_received

```
uint64_t obsws_stats_t::messages_received
```

Definition at line 285 of file library.h.

3.4.2.7 messages_sent

```
uint64_t obsws_stats_t::messages_sent
```

Definition at line 284 of file library.h.

3.4.2.8 reconnect_count

```
uint64_t obsws_stats_t::reconnect_count
```

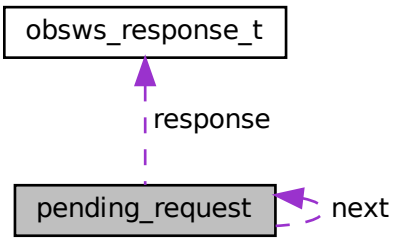
Definition at line 288 of file library.h.

The documentation for this struct was generated from the following file:

- library.h

3.5 pending_request Struct Reference

Collaboration diagram for pending_request:



Data Fields

- char request_id [OBSWS_UUID_LENGTH]
- obsws_response_t * response
- bool completed
- pthread_mutex_t mutex
- pthread_cond_t cond
- time_t timestamp
- struct pending_request * next

3.5.1 Detailed Description

Definition at line 127 of file library.c.

3.5.2 Field Documentation

3.5.2.1 completed

```
bool pending_request::completed
```

Definition at line 130 of file library.c.

3.5.2.2 cond

```
pthread_cond_t pending_request::cond
```

Definition at line 132 of file library.c.

3.5.2.3 mutex

```
pthread_mutex_t pending_request::mutex
```

Definition at line 131 of file library.c.

3.5.2.4 next

```
struct pending_request* pending_request::next
```

Definition at line 134 of file library.c.

3.5.2.5 request_id

```
char pending_request::request_id[OBSWS_UUID_LENGTH]
```

Definition at line 128 of file library.c.

3.5.2.6 response

```
obsws_response_t* pending_request::response
```

Definition at line 129 of file library.c.

3.5.2.7 timestamp

```
time_t pending_request::timestamp
```

Definition at line 133 of file library.c.

The documentation for this struct was generated from the following file:

- library.c

Chapter 4

File Documentation

4.1 library.c File Reference

```
#include "library.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <pthread.h>
#include <unistd.h>
#include <stdarg.h>
#include <sys/socket.h>
#include <sys/select.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <errno.h>
#include <poll.h>
#include <libwebsockets.h>
#include <openssl/sha.h>
#include <openssl/evp.h>
#include <openssl/bio.h>
#include <openssl/buffer.h>
#include <cjson/cJSON.h>
```

Include dependency graph for library.c:

Data Structures

- struct pending_request
- struct obsws_connection

Macros

- #define _POSIX_C_SOURCE 200809L
- #define OBSWS_VERSION "1.0.0" /* Library version string */
- #define OBSWS_PROTOCOL_VERSION 1 /* OBS WebSocket protocol version (v5 uses RPC version 1) */
- #define OBSWS_DEFAULT_BUFFER_SIZE 65536 /* 64KB buffer for WebSocket messages */
- #define OBSWS_MAX_PENDING_REQUESTS 256
- #define OBSWS_UUID_LENGTH 37
- #define OBSWS_OPCODE_HELLO 0 /* Server: Initial greeting with auth info */
- #define OBSWS_OPCODE_IDENTIFY 1 /* Client: Authentication and protocol agreement */
- #define OBSWS_OPCODE_IDENTIFIED 2 /* Server: Auth successful, ready for commands */
- #define OBSWS_OPCODE_REIDENTIFY 3 /* Client: Re-authenticate after reconnect */
- #define OBSWS_OPCODE_EVENT 5 /* Server: Something happened in OBS */
- #define OBSWS_OPCODE_REQUEST 6 /* Client: Execute an operation in OBS */
- #define OBSWS_OPCODE_REQUEST_RESPONSE 7 /* Server: Result of a client request */
- #define OBSWS_OPCODE_REQUEST_BATCH 8 /* Client: Multiple requests at once (unused) */
- #define OBSWS_OPCODE_REQUEST_BATCH_RESPONSE 9 /* Server: Responses to batch (unused) */
- #define OBSWS_EVENT_GENERAL (1 << 0) /* General OBS events (startup, shutdown) */
- #define OBSWS_EVENT_CONFIG (1 << 1) /* Configuration change events */
- #define OBSWS_EVENT_SCENES (1 << 2) /* Scene-related events (scene switched, etc) */
- #define OBSWS_EVENT_INPUTS (1 << 3) /* Input source events (muted, volume changed) */
- #define OBSWS_EVENT_TRANSITIONS (1 << 4) /* Transition events (transition started) */
- #define OBSWS_EVENT_FILTERS (1 << 5) /* Filter events (filter added, removed) */
- #define OBSWS_EVENT_OUTPUTS (1 << 6) /* Output events (recording started, streaming stopped) */
- #define OBSWS_EVENT_SCENE_ITEMS (1 << 7) /* Scene item events (source added to scene) */
- #define OBSWS_EVENT_MEDIA_INPUTS (1 << 8) /* Media playback events (media finished) */
- #define OBSWS_EVENT_VENDORS (1 << 9) /* Vendor-specific extensions */
- #define OBSWS_EVENT_UI (1 << 10) /* UI events (Studio Mode toggled) */
- #define OBSWS_EVENT_ALL 0x7FF /* Subscribe to all event types */

Typedefs

- typedef struct pending_request pending_request_t

Functions

- static void obsws_log (obsws_connection_t *conn, obsws_log_level_t level, const char *format,...)
- static void obsws_debug (obsws_connection_t *conn, obsws_debug_level_t min_level, const char *format,...)
- static void generate_uuid (char *uuid_out)
- static char * base64_encode (const unsigned char *input, size_t length)
- static void sha256_hash (const char *input, unsigned char *output)
- static char * generate_auth_response (const char *password, const char *salt, const char *challenge)
- static void set_connection_state (obsws_connection_t *conn, obsws_state_t new_state)
- static pending_request_t * create_pending_request (obsws_connection_t *conn, const char *request_id)
- static pending_request_t * find_pending_request (obsws_connection_t *conn, const char *request_id)
- static void remove_pending_request (obsws_connection_t *conn, pending_request_t *target)
- static void cleanup_old_requests (obsws_connection_t *conn)
- static int handle_hello_message (obsws_connection_t *conn, cJSON *data)

  Handle the initial HELLO handshake message from OBS.
- static int handle_identified_message (obsws_connection_t *conn, cJSON *data)

  Handle the IDENTIFIED confirmation message from OBS.
- static int handle_event_message (obsws_connection_t *conn, cJSON *data)

Handle EVENT messages from OBS (real-time notifications).

- static int handle_request_response_message (obsws_connection_t *conn, cJSON *data)

  Handle REQUEST_RESPONSE messages from OBS (responses to our commands).
- static int handle_websocket_message (obsws_connection_t *conn, const char *message, size_t len)

  Route incoming WebSocket messages to appropriate handlers based on opcode.
- static int lws_callback (struct lws *wsi, enum lws_callback_reasons reason, void *user, void *in, size_t len)

  libwebsockets callback - routes WebSocket events to our handlers.
- static void * event_thread_func (void *arg)

  Background thread function that continuously processes WebSocket events.
- obsws_error_t obsws_init (void)

  Initialize the libwsv5 library.
- void obsws_cleanup (void)

  Clean up library resources.
- const char * obsws_version (void)

  Get the library version string.
- void obsws_set_log_level (obsws_log_level_t level)

  Set the global log level threshold.
- void obsws_set_debug_level (obsws_debug_level_t level)

  Set the global debug level.
- obsws_debug_level_t obsws_get_debug_level (void)

  Get the current debug level.
- void obsws_config_init (obsws_config_t *config)

  Initialize a connection configuration structure with safe defaults.
- obsws_connection_t * obsws_connect (const obsws_config_t *config)

  Establish a connection to OBS.
- void obsws_disconnect (obsws_connection_t *conn)

  Disconnect from OBS and clean up connection resources.
- bool obsws_is_connected (const obsws_connection_t *conn)

  Check if a connection is actively connected to OBS.
- obsws_state_t obsws_get_state (const obsws_connection_t *conn)

  Get the current connection state.
- obsws_error_t obsws_get_stats (const obsws_connection_t *conn, obsws_stats_t *stats)

  Retrieve performance and connectivity statistics.
- obsws_error_t obsws_send_request (obsws_connection_t *conn, const char *request_type, const char *request_data, obsws_response_t **response, uint32_t timeout_ms)

  Send a synchronous request to OBS and wait for the response.
- obsws_error_t obsws_set_current_scene (obsws_connection_t *conn, const char *scene_name, obsws_response_t **response)

  Switch OBS to a specific scene.
- obsws_error_t obsws_get_current_scene (obsws_connection_t *conn, char *scene_name, size_t buffer_↩ size)

  Query the currently active scene in OBS.
- obsws_error_t obsws_start_recording (obsws_connection_t *conn, obsws_response_t **response)

  Start recording in OBS.
- obsws_error_t obsws_stop_recording (obsws_connection_t *conn, obsws_response_t **response)

  Stop recording in OBS.
- obsws_error_t obsws_start_streaming (obsws_connection_t *conn, obsws_response_t **response)

  Start streaming in OBS.
- obsws_error_t obsws_stop_streaming (obsws_connection_t *conn, obsws_response_t **response)

  Stop streaming in OBS.
- void obsws_response_free (obsws_response_t *response)

Free a response object previously allocated by obsws_send_request().

- const char * obsws_error_string (obsws_error_t error)

  Convert an error code to a human-readable string.

- const char * obsws_state_string (obsws_state_t state)

  Convert a connection state to a human-readable string.

- int obsws_process_events (obsws_connection_t *conn, uint32_t timeout_ms)

  Process pending WebSocket events (compatibility function).

Variables

- static bool g_library_initialized = false
- static obsws_log_level_t g_log_level = OBSWS_LOG_INFO
- static obsws_debug_level_t g_debug_level = OBSWS_DEBUG_NONE
- static pthread_mutex_t g_init_mutex = PTHREAD_MUTEX_INITIALIZER
- static const struct lws_protocols protocols [ ]

4.1.1 Macro Definition Documentation

4.1.1.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

Definition at line 1 of file library.c.

4.1.1.2 OBSWS_DEFAULT_BUFFER_SIZE

```
#define OBSWS_DEFAULT_BUFFER_SIZE 65536 /* 64KB buffer for WebSocket messages */
```

Definition at line 39 of file library.c.

4.1.1.3 OBSWS_EVENT_ALL

```
#define OBSWS_EVENT_ALL 0x7FF /* Subscribe to all event types */
```

Definition at line 101 of file library.c.

4.1.1.4 OBSWS_EVENT_CONFIG

```
#define OBSWS_EVENT_CONFIG (1 << 1) /* Configuration change events */
```

Definition at line 91 of file library.c.

4.1.1.5 OBSWS_EVENT_FILTERS

```
#define OBSWS_EVENT_FILTERS (1 << 5) /* Filter events (filter added, removed) */
```

Definition at line 95 of file library.c.

4.1.1.6 OBSWS_EVENT_GENERAL

```
#define OBSWS_EVENT_GENERAL (1 << 0) /* General OBS events (startup, shutdown) */
```

Definition at line 90 of file library.c.

4.1.1.7 OBSWS_EVENT_INPUTS

```
#define OBSWS_EVENT_INPUTS (1 << 3) /* Input source events (muted, volume changed) */
```

Definition at line 93 of file library.c.

4.1.1.8 OBSWS_EVENT_MEDIA_INPUTS

```
#define OBSWS_EVENT_MEDIA_INPUTS (1 << 8) /* Media playback events (media finished) */
```

Definition at line 98 of file library.c.

4.1.1.9 OBSWS_EVENT_OUTPUTS

```
#define OBSWS_EVENT_OUTPUTS (1 << 6) /* Output events (recording started, streaming stopped)
*/
```

Definition at line 96 of file library.c.

4.1.1.10 OBSWS_EVENT_SCENE_ITEMS

```
#define OBSWS_EVENT_SCENE_ITEMS (1 << 7) /* Scene item events (source added to scene) */
```

Definition at line 97 of file library.c.

4.1.1.11 OBSWS_EVENT_SCENES

```
#define OBSWS_EVENT_SCENES (1 << 2) /* Scene-related events (scene switched, etc) */
```

Definition at line 92 of file library.c.

4.1.1.12 OBSWS_EVENT_TRANSITIONS

```
#define OBSWS_EVENT_TRANSITIONS (1 << 4) /* Transition events (transition started) */
```

Definition at line 94 of file library.c.

4.1.1.13 OBSWS_EVENT_UI

```
#define OBSWS_EVENT_UI (1 << 10) /* UI events (Studio Mode toggled) */
```

Definition at line 100 of file library.c.

4.1.1.14 OBSWS_EVENT_VENDORS

```
#define OBSWS_EVENT_VENDORS (1 << 9) /* Vendor-specific extensions */
```

Definition at line 99 of file library.c.

4.1.1.15 OBSWS_MAX_PENDING_REQUESTS

```
#define OBSWS_MAX_PENDING_REQUESTS 256
```

Definition at line 45 of file library.c.

4.1.1.16 OBSWS_OPCODE_EVENT

```
#define OBSWS_OPCODE_EVENT 5 /* Server:  Something happened in OBS */
```

Definition at line 72 of file library.c.

4.1.1.17 OBSWS_OPCODE_HELLO

```
#define OBSWS_OPCODE_HELLO 0 /* Server:  Initial greeting with auth info */
```

Definition at line 68 of file library.c.

4.1.1.18 OBSWS_OPCODE_IDENTIFIED

```
#define OBSWS_OPCODE_IDENTIFIED 2 /* Server:  Auth successful, ready for commands */
```

Definition at line 70 of file library.c.

4.1.1.19 OBSWS_OPCODE_IDENTIFY

```
#define OBSWS_OPCODE_IDENTIFY 1 /* Client:  Authentication and protocol agreement */
```

Definition at line 69 of file library.c.

4.1.1.20 OBSWS_OPCODE_REIDENTIFY

```
#define OBSWS_OPCODE_REIDENTIFY 3 /* Client:  Re-authenticate after reconnect */
```

Definition at line 71 of file library.c.

4.1.1.21 OBSWS_OPCODE_REQUEST

```
#define OBSWS_OPCODE_REQUEST 6 /* Client:  Execute an operation in OBS */
```

Definition at line 73 of file library.c.

4.1.1.22 OBSWS_OPCODE_REQUEST_BATCH

```
#define OBSWS_OPCODE_REQUEST_BATCH 8 /* Client:  Multiple requests at once (unused) */
```

Definition at line 75 of file library.c.

4.1.1.23 OBSWS_OPCODE_REQUEST_BATCH_RESPONSE

```
#define OBSWS_OPCODE_REQUEST_BATCH_RESPONSE 9 /* Server:  Responses to batch (unused) */
```

Definition at line 76 of file library.c.

4.1.1.24 OBSWS_OPCODE_REQUEST_RESPONSE

```
#define OBSWS_OPCODE_REQUEST_RESPONSE 7 /* Server:  Result of a client request */
```

Definition at line 74 of file library.c.

4.1.1.25 OBSWS_PROTOCOL_VERSION

```
#define OBSWS_PROTOCOL_VERSION 1 /* OBS WebSocket protocol version (v5 uses RPC version 1) */
```

Definition at line 33 of file library.c.

4.1.1.26 OBSWS_UUID_LENGTH

```
#define OBSWS_UUID_LENGTH 37
```

Definition at line 49 of file library.c.

4.1.1.27 OBSWS_VERSION

```
#define OBSWS_VERSION "1.0.0" /* Library version string */
```

Definition at line 32 of file library.c.

4.1.2 Typedef Documentation

4.1.2.1 pending_request_t

```
typedef struct pending_request pending_request_t
```

4.1.3 Function Documentation

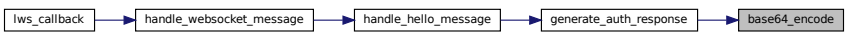
4.1.3.1 base64_encode()

```
static char * base64_encode (
            const unsigned char * input,
            size_t length )  [static]
```

Definition at line 369 of file library.c.

```
00369                                                                   {
00370     BIO *bio, *b64;
00371     BUF_MEM *buffer_ptr;
00372
00373     /* Set up OpenSSL base64 encoder:  b64 filter pushing to memory buffer */
00374     b64 = BIO_new(BIO_f_base64());
00375     bio = BIO_new(BIO_s_mem());
00376     bio = BIO_push(b64, bio);
00377
00378     /* Disable newlines in output (OpenSSL adds them by default for readability) */
00379     BIO_set_flags(bio, BIO_FLAGS_BASE64_NO_NL);
00380
00381     /* Encode the data */
00382     BIO_write(bio, input, length);
00383     BIO_flush(bio);
00384     BIO_get_mem_ptr(bio, &buffer_ptr);
00385
00386     /* Copy result to our own allocated buffer and null-terminate */
00387     char *result = malloc(buffer_ptr->length + 1);
00388     memcpy(result, buffer_ptr->data, buffer_ptr->length);
00389     result[buffer_ptr->length] = '\0';
00390
00391     BIO_free_all(bio);
00392     return result;
00393 }
```

Here is the caller graph for this function:



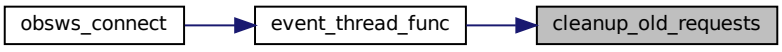
4.1.3.2 cleanup_old_requests()

```
static void cleanup_old_requests (
            obsws_connection_t * conn )  [static]
```

Definition at line 572 of file library.c.

```
00572                                                                    {
00573     time_t now = time(NULL);
00574     pthread_mutex_lock(&conn->requests_mutex);
00575
00576     pending_request_t **req = &conn->pending_requests;
00577     while (*req) {
00578         /* Check if request has timed out (30 seconds) */
00579         if (now - (*req)->timestamp > 30) {
00580             pending_request_t *old = *req;
00581             *req = old->next;
00582
00583             /* Mark as completed with timeout error */
00584             pthread_mutex_lock(&old->mutex);
00585             old->completed = true;
00586             old->response->success = false;
00587             old->response->error_message = strdup("Request timeout");
00588             pthread_cond_broadcast(&old->cond);  /* Wake waiting threads */
00589             pthread_mutex_unlock(&old->mutex);
00590         } else {
00591             req = &(*req)->next;
00592         }
00593     }
00594
00595     pthread_mutex_unlock(&conn->requests_mutex);
00596 }
```

Here is the caller graph for this function:



4.1.3.3 create_pending_request()

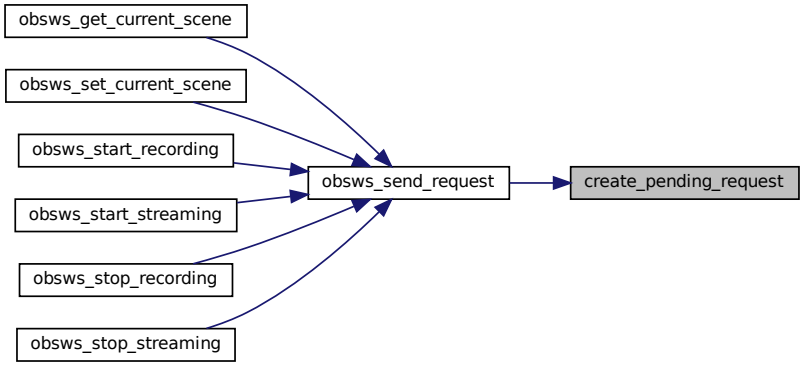
```
static pending_request_t * create_pending_request (
            obsws_connection_t * conn,
            const char * request_id )  [static]
```

Definition at line 506 of file library.c.

```
00506                                                                                                                {
00507     pending_request_t *req = calloc(1, sizeof(pending_request_t));
00508     if (!req) return NULL;
00509
00510     /* Copy request ID and ensure null termination */
00511     strncpy(req->request_id, request_id, OBSWS_UUID_LENGTH - 1);
00512     req->request_id[OBSWS_UUID_LENGTH - 1] = '\0';
00513
00514     /* Initialize request structure */
00515     req->response = calloc(1, sizeof(obsws_response_t));
00516     req->completed = false;
00517     req->timestamp = time(NULL);
00518     pthread_mutex_init(&req->mutex, NULL);
00519     pthread_cond_init(&req->cond, NULL);
00520
00521     /* Add to linked list of pending requests */
00522     pthread_mutex_lock(&conn->requests_mutex);
00523     req->next = conn->pending_requests;
00524     conn->pending_requests = req;
```

```
00525     pthread_mutex_unlock(&conn->requests_mutex);
00526
00527     return req;
00528 }
```

Here is the caller graph for this function:



4.1.3.4 event_thread_func()

```
static void * event_thread_func (
            void * arg )  [static]
```

Background thread function that continuously processes WebSocket events.

Each connection has one background thread dedicated to processing WebSocket messages and timers. The main application thread remains free to make requests and do application work without blocking.

This thread:

1. Calls lws_service() to pump the libwebsockets event loop (typically blocks for 50ms waiting for events, then processes them and returns)
2. Periodically cleans up old/timed-out requests
3. Sends keep-alive pings if configured (to detect dead connections)
4. Exits gracefully when should_exit flag is set

Lifetime: This thread is created in obsws_connect() and destroyed in obsws_disconnect() using pthread_join(). The should_exit flag is used to signal the thread to stop, which it checks at the start of each loop.

The lws_service() call is the core of this loop. It:

- Waits up to 50ms for data from the network using select/poll
- If data arrives, invokes lws_callback to notify us

- Returns after ~50ms even if no data (so we stay responsive to should_exit)

This asynchronous design has several advantages:

- App thread isn't blocked waiting for responses
- Multiple requests can be in-flight simultaneously
- Events can be delivered to callbacks instantly (no polling delay)
- Automatic keep-alive pings keep the connection alive through firewalls

Memory note: Callbacks invoked from this thread have access to the same connection object as the main thread, hence all the mutexes protecting critical sections. The pending_request_t condition variables synchronize request responses between this thread and application threads.

Parameters

arg	The obsws_connection_t* that this thread services

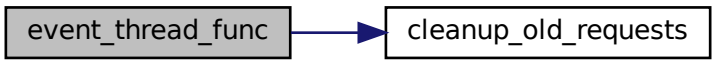
Returns

Always NULL (threads don't return values)

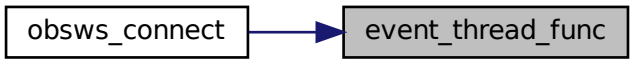
Definition at line 1131 of file library.c.

```
01131                                       {
01132     obsws_connection_t *conn = (obsws_connection_t *)arg;
01133
01134     bool should_continue = true;
01135     while (should_continue) {
01136         /* Check exit flag with mutex protection */
01137         pthread_mutex_lock(&conn->state_mutex);
01138         should_continue = !conn->should_exit;
01139         pthread_mutex_unlock(&conn->state_mutex);
01140
01141         if (!should_continue) break;
01142
01143         if (conn->lws_context) {
01144             lws_service(conn->lws_context, 50);
01145
01146             /* Cleanup old requests periodically */
01147             cleanup_old_requests(conn);
01148
01149             /* Handle keep-alive pings */
01150             if (conn->config.ping_interval_ms > 0 && conn->state == OBSWS_STATE_CONNECTED) {
01151                 time_t now = time(NULL);
01152                 if (now - conn->last_ping_sent >= conn->config.ping_interval_ms / 1000) {
01153                     if (conn->wsi) {
01154                         lws_callback_on_writable(conn->wsi);
01155                     }
01156                     conn->last_ping_sent = now;
01157                 }
01158             }
01159         } else {
01160             struct timespec ts = {0, 50000000};  /* 50ms = 50,000,000 nanoseconds */
01161             nanosleep(&ts, NULL);
01162         }
01163     }
01164
01165     return NULL;
01166 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



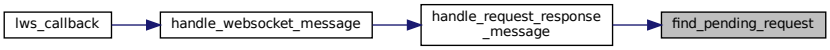
4.1.3.5 find_pending_request()

```
static pending_request_t * find_pending_request (
            obsws_connection_t * conn,
            const char * request_id )  [static]
```

Definition at line 531 of file library.c.

```
00531                                                                                                {
00532     pthread_mutex_lock(&conn->requests_mutex);
00533     pending_request_t *req = conn->pending_requests;
00534
00535     /* Search linked list for matching request ID */
00536     while (req) {
00537         if (strcmp(req->request_id, request_id) == 0) {
00538             pthread_mutex_unlock(&conn->requests_mutex);
00539             return req;
00540         }
00541         req = req->next;
00542     }
00543
00544     pthread_mutex_unlock(&conn->requests_mutex);
00545     return NULL;
00546 }
```

Here is the caller graph for this function:



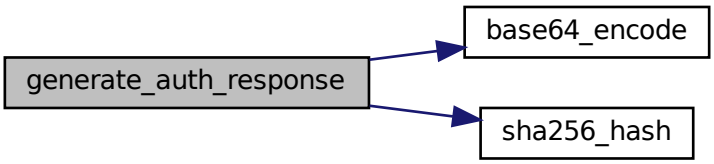
4.1.3.6 generate_auth_response()

```
static char * generate_auth_response (
            const char * password,
            const char * salt,
            const char * challenge )  [static]
```

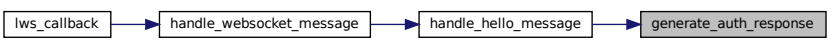
Definition at line 437 of file library.c.

```
00437                                                                                {
00438     unsigned char secret_hash[SHA256_DIGEST_LENGTH];
00439     unsigned char auth_hash[SHA256_DIGEST_LENGTH];
00440
00441     /* Step 1:  Compute secret = base64(sha256(password + salt)) */
00442     char *password_salt = malloc(strlen(password) + strlen(salt) + 1);
00443     sprintf(password_salt, "%s%s", password, salt);
00444     sha256_hash(password_salt, secret_hash);
00445     free(password_salt);
00446
00447     char *secret = base64_encode(secret_hash, SHA256_DIGEST_LENGTH);
00448
00449     /* Step 2:  Compute auth response = base64(sha256(secret + challenge)) */
00450     char *secret_challenge = malloc(strlen(secret) + strlen(challenge) + 1);
00451     sprintf(secret_challenge, "%s%s", secret, challenge);
00452     sha256_hash(secret_challenge, auth_hash);
00453     free(secret_challenge);
00454     free(secret);
00455
00456     /* Return the final response, base64-encoded */
00457     char *auth_response = base64_encode(auth_hash, SHA256_DIGEST_LENGTH);
00458     return auth_response;
00459 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



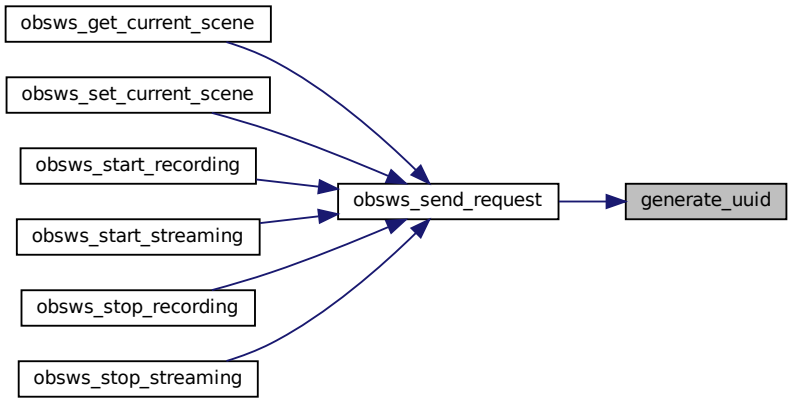
4.1.3.7 generate_uuid()

```
static void generate_uuid (
            char * uuid_out )  [static]
```

Definition at line 344 of file library.c.

```
00344                                          {
00345     unsigned int r1 = rand();
00346     unsigned int r2 = rand();
00347     unsigned int r3 = rand();
00348     unsigned int r4 = rand();
00349
00350     sprintf(uuid_out, "%08x-%04x-%04x-%04x-%04x%08x",
00351             r1,                             /* 8 hex digits */
00352             r2 & 0xFFFF,                   /* 4 hex digits */
00353             (r3 & 0x0FFF) | 0x4000,        /* 4 hex digits (set version 4) */
00354             (r4 & 0x3FFF) | 0x8000,        /* 4 hex digits (set variant bits) */
00355             rand() & 0xFFFF,               /* 4 hex digits */
00356             (unsigned int)rand());          /* 8 hex digits */
00357 }
```

Here is the caller graph for this function:



4.1.3.8 handle_event_message()

```
static int handle_event_message (
            obsws_connection_t * conn,
            cJSON * data )  [static]
```

Handle EVENT messages from OBS (real-time notifications).

OBS continuously sends EVENT messages whenever something happens in the application (scene changes, source muted/unmuted, recording started, etc.). These events are only delivered if we subscribed to them in the IDENTIFY message using the eventSubscriptions bitmask.

This function:

1. Extracts the event type and event data from the JSON
2. Calls the user's event_callback if one was configured (async event loop)
3. Updates internal caches (e.g., current scene name on SceneChanged)

Important threading note: This is called from the background event_thread, NOT from the main application thread. Therefore:

- The event_callback is executed in the event thread context
- The callback should not block (keep processing fast)
- The callback should not make blocking library calls
- The event_data_str parameter is temporary and freed after callback returns
- If the callback needs to keep the data, it must copy it

Scene caching optimization: We maintain a cache of the currently active scene name in the connection structure. When we see a CurrentProgramSceneChanged event, we update this cache immediately. This avoids the need for the application to call obsws_send_request(..., "GetCurrentProgramScene", ...) repeatedly. The cache is protected by scene_mutex for thread safety.

Parameters

conn	The connection that received the event
data	The parsed JSON "d" field containing eventType and eventData

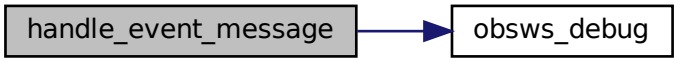
Returns

0 on success, -1 if the event data is malformed

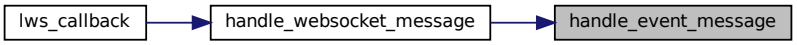
Definition at line 786 of file library.c.

```
00786                                                                   {
00787     cJSON *event_type = cJSON_GetObjectItem(data, "eventType");
00788     cJSON *event_data = cJSON_GetObjectItem(data, "eventData");
00789
00790     /* DEBUG_MEDIUM: Show event type */
00791     if (event_type) {
00792         obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Event received:  %s", event_type->valuestring);
00793     }
00794
00795     if (event_type && conn->config.event_callback) {
00796         char *event_data_str = event_data ?  cJSON_PrintUnformatted(event_data) :  NULL;
00797         /* DEBUG_HIGH: Show full event data */
00798         if (event_data_str) {
00799             obsws_debug(conn, OBSWS_DEBUG_HIGH, "Event data:  %s", event_data_str);
00800         }
00801         conn->config.event_callback(conn, event_type->valuestring, event_data_str,
      conn->config.user_data);
00802         if (event_data_str) free(event_data_str);
00803     }
00804
00805     /* Update current scene cache if scene changed */
00806     if (event_type && strcmp(event_type->valuestring, "CurrentProgramSceneChanged") == 0) {
00807         cJSON *scene_name = cJSON_GetObjectItem(event_data, "sceneName");
00808         if (scene_name) {
00809             pthread_mutex_lock(&conn->scene_mutex);
00810             free(conn->current_scene);
00811             conn->current_scene = strdup(scene_name->valuestring);
00812             pthread_mutex_unlock(&conn->scene_mutex);
00813             /* DEBUG_LOW: Scene changes are important */
00814             obsws_debug(conn, OBSWS_DEBUG_LOW, "Scene changed to:  %s", scene_name->valuestring);
00815         }
00816     }
00817
00818     return 0;
00819 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.3.9 handle_hello_message()

```
static int handle_hello_message (
            obsws_connection_t * conn,
            cJSON * data )  [static]
```

Handle the initial HELLO handshake message from OBS.

When we first connect to OBS, the server sends a HELLO message containing protocol version information and, if required, an authentication challenge and salt. This function extracts that information and immediately responds with an IDENTIFY message.

The authentication flow (if enabled) works as follows:

1. Server sends HELLO with a random challenge string and a salt
2. We compute: secret = base64(SHA256(password + salt))
3. We compute: response = base64(SHA256(secret + challenge))
4. We send this response in the IDENTIFY message
5. If it matches what the server computed, auth succeeds

This challenge-response approach has several advantages over sending the raw password:

- Password never travels across the network (only the computed response)
- If someone captures the network traffic, they can't replay the captured response because it's specific to this challenge (which was random)
- Similar to HTTP Digest Authentication (RFC 2617) but simpler

The function transitions the connection state from CONNECTING to AUTHENTICATING, sends the IDENTIFY message, and logs any authentication requirements.

Parameters

conn	The connection structure containing buffers, config, and state
data	The parsed JSON "d" field from the HELLO message, containing authentication info if auth is required

Returns

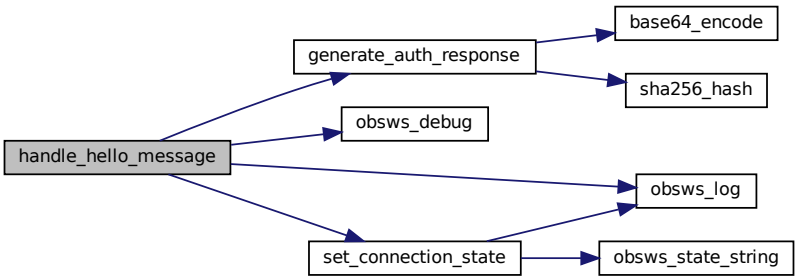
0 on success, -1 on error (though errors are logged and connection continues - failure to authenticate will be detected by the server refusing to transition to IDENTIFIED state)

Definition at line 636 of file library.c.

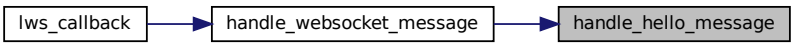
```
00636                                                                   {
00637     /* DEBUG_LOW: Basic connection event */
00638     obsws_debug(conn, OBSWS_DEBUG_LOW, "Received Hello message from OBS");
00639 
00640     cJSON *auth = cJSON_GetObjectItem(data, "authentication");
00641     if (auth) {
00642         conn->auth_required = true;
00643 
00644         cJSON *challenge = cJSON_GetObjectItem(auth, "challenge");
00645         cJSON *salt = cJSON_GetObjectItem(auth, "salt");
00646 
00647         if (challenge && salt) {
00648             conn->challenge = strdup(challenge->valuestring);
00649             conn->salt = strdup(salt->valuestring);
00650             /* DEBUG_MEDIUM: Show auth parameters */
00651             obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Authentication required - salt:  %s, challenge:
    %s",
00652                         conn->salt, conn->challenge);
00653         }
00654     } else {
00655         conn->auth_required = false;
00656         obsws_debug(conn, OBSWS_DEBUG_LOW, "No authentication required");
00657     }
00658 
00659     /* Send Identify message */
00660     set_connection_state(conn, OBSWS_STATE_AUTHENTICATING);
00661 
00662     cJSON *identify = cJSON_CreateObject();
00663     cJSON_AddNumberToObject(identify, "op", OBSWS_OPCODE_IDENTIFY);
00664 
00665     cJSON *identify_data = cJSON_CreateObject();
00666     cJSON_AddNumberToObject(identify_data, "rpcVersion", OBSWS_PROTOCOL_VERSION);
00667     cJSON_AddNumberToObject(identify_data, "eventSubscriptions", OBSWS_EVENT_ALL);
00668 
00669     if (conn->auth_required && conn->config.password) {
00670         /* DEBUG_HIGH: Show password being used */
00671         obsws_debug(conn, OBSWS_DEBUG_HIGH, "Generating auth response with password:  '%s'",
    conn->config.password);
00672         char *auth_response = generate_auth_response(conn->config.password, conn->salt,
    conn->challenge);
00673         /* DEBUG_MEDIUM: Show generated auth string */
00674         obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Generated auth response:  '%s'", auth_response);
00675         cJSON_AddStringToObject(identify_data, "authentication", auth_response);
00676         free(auth_response);
00677     } else {
00678         if (conn->auth_required) {
00679             obsws_log(conn, OBSWS_LOG_ERROR, "Authentication required but no password provided!");
00680         }
00681     }
00682 
00683     cJSON_AddItemToObject(identify, "d", identify_data);
00684 
00685     char *message = cJSON_PrintUnformatted(identify);
00686     cJSON_Delete(identify);
00687 
00688     /* DEBUG_HIGH: Show full Identify message */
00689     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending Identify message:  %s", message);
00690 
00691     pthread_mutex_lock(&conn->send_mutex);
00692     size_t len = strlen(message);
00693     if (len < conn->send_buffer_size - LWS_PRE) {
00694         memcpy(conn->send_buffer + LWS_PRE, message, len);
00695         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
    LWS_WRITE_TEXT);
00696         /* DEBUG_HIGH: Show bytes sent */
00697         obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sent %d bytes (requested %zu)", written, len);
00698     } else {
00699         obsws_log(conn, OBSWS_LOG_ERROR, "Message too large for send buffer:  %zu bytes", len);
```

```
00700     }
00701     pthread_mutex_unlock(&conn->send_mutex);
00702
00703     free(message);
00704     return 0;
00705 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.3.10 handle_identified_message()

```
static int handle_identified_message (
            obsws_connection_t * conn,
            cJSON * data )  [static]
```

Handle the IDENTIFIED confirmation message from OBS.

After we send an IDENTIFY message with authentication (or without if auth isn't required), OBS responds with an IDENTIFIED message to confirm that the connection is established and ready for commands. This function marks the connection as fully established, records connection statistics, and resets the reconnection state.

Receiving this message means:

- Authentication succeeded (if it was required)
- The server has accepted our protocol version
- We can now send REQUEST messages and receive EVENT messages
- The connection is in a healthy state

We take this opportunity to:

1. Log successful authentication
2. Transition state to CONNECTED (the only valid way to enter this state)
3. Record the timestamp of successful connection (for statistics)
4. Reset the reconnection attempt counter and delay (we're connected!)

Parameters

conn	The connection structure to mark as identified
data	Unused (the IDENTIFIED message typically has no data payload)

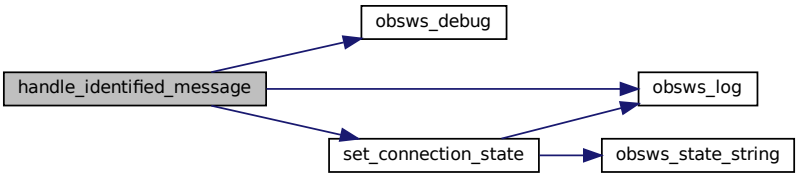
Returns

Always 0 (this message type should never fail)

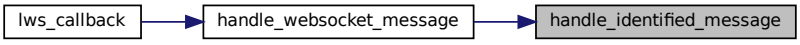
Definition at line 734 of file library.c.

```
00734                                                                    {
00735     (void)data;  /* Unused parameter */
00736     obsws_log(conn, OBSWS_LOG_INFO, "Successfully authenticated with OBS");
00737     /* DEBUG_LOW: Authentication success */
00738     obsws_debug(conn, OBSWS_DEBUG_LOW, "Identified message received - authentication successful");
00739     set_connection_state(conn, OBSWS_STATE_CONNECTED);
00740 
00741     pthread_mutex_lock(&conn->stats_mutex);
00742     conn->stats.connected_since = time(NULL);
00743     conn->stats.reconnect_count = conn->reconnect_attempts;
00744     pthread_mutex_unlock(&conn->stats_mutex);
00745 
00746     conn->reconnect_attempts = 0;
00747     conn->current_reconnect_delay = conn->config.reconnect_delay_ms;
00748 
00749     return 0;
00750 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.3.11 handle_request_response_message()

```
static int handle_request_response_message (
            obsws_connection_t * conn,
            cJSON * data )  [static]
```

Handle REQUEST_RESPONSE messages from OBS (responses to our commands).

When we send a REQUEST message (via obsws_send_request), OBS processes it and sends back a REQUEST↩ _RESPONSE message with the same requestId that we used. This function matches the response to the pending request, populates the response data, and wakes up the waiting thread.

The async request/response pattern allows the application to send multiple requests without waiting for each response. The flow is:

1. Application calls obsws_send_request("GetScenes", ...) -> returns immediately
2. The request is created with a unique UUID and added to pending_requests list
3. Background thread sends the request to OBS
4. Background thread waits for response (on condition variable, not busy-polling)
5. OBS responds with REQUEST_RESPONSE containing the requestId
6. This function matches it to the pending request
7. Function sets response->success, response->status_code, response->response_data
8. Function signals the condition variable to wake the waiting thread
9. Application thread wakes up with the response ready

This is far more efficient than synchronous request/response because:

- No blocking wait in the background thread
- Multiple requests can be in-flight simultaneously
- Doesn't freeze the application while waiting for OBS
- Condition variables are more efficient than polling

Response structure contains:

- success: Did the operation succeed? (not the HTTP status, but "was it valid?")
- status_code: The OBS response code (0 = success, >0 = error)
- response_data: JSON string with the actual result (e.g., scene list)
- error_message: If something failed, what was the reason?

Parameters

conn	The connection that received the response
data	The parsed JSON "d" field containing requestId, requestStatus, etc.

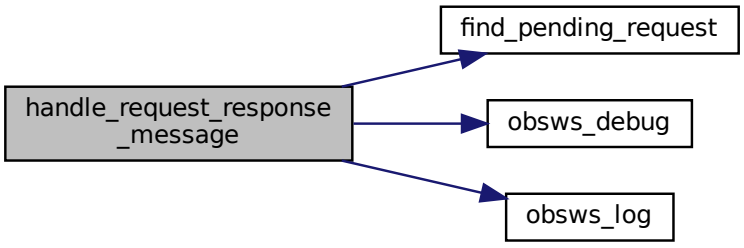
Returns

0 on success, -1 if the response is malformed (e.g., missing requestId)

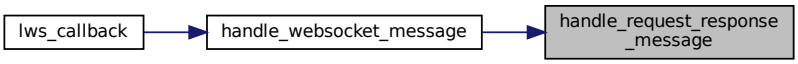
Definition at line 859 of file library.c.

```
00859                                                                                {
00860     cJSON *request_id = cJSON_GetObjectItem(data, "requestId");
00861     if (!request_id) return -1;
00862
00863     /* DEBUG_MEDIUM: Show request ID being processed */
00864     obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Response received for request:  %s",
      request_id->valuestring);
00865
00866     pending_request_t *req = find_pending_request(conn, request_id->valuestring);
00867     if (!req) {
00868         obsws_log(conn, OBSWS_LOG_WARNING, "Received response for unknown request:  %s",
      request_id->valuestring);
00869         return -1;
00870     }
00871
00872     pthread_mutex_lock(&req->mutex);
00873
00874     cJSON *request_status = cJSON_GetObjectItem(data, "requestStatus");
00875     if (request_status) {
00876         cJSON *result = cJSON_GetObjectItem(request_status, "result");
00877         cJSON *code = cJSON_GetObjectItem(request_status, "code");
00878         cJSON *comment = cJSON_GetObjectItem(request_status, "comment");
00879
00880         req->response->success = result ?  result->valueint :  false;
00881         req->response->status_code = code ?  code->valueint :  -1;
00882
00883         if (comment) {
00884             req->response->error_message = strdup(comment->valuestring);
00885         }
00886     }
00887
00888     cJSON *response_data = cJSON_GetObjectItem(data, "responseData");
00889     if (response_data) {
00890         req->response->response_data = cJSON_PrintUnformatted(response_data);
00891     }
00892
00893     req->completed = true;
00894     pthread_cond_broadcast(&req->cond);
00895     pthread_mutex_unlock(&req->mutex);
00896
00897     return 0;
00898 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.3.12 handle_websocket_message()

```
static int handle_websocket_message (
            obsws_connection_t * conn,
            const char * message,
            size_t len )  [static]
```

Route incoming WebSocket messages to appropriate handlers based on opcode.

Every message from OBS contains an "op" field (opcode) that identifies the message type. This function:

1. Parses the JSON to extract the opcode and data
2. Routes to the appropriate handler function based on the opcode
3. Updates statistics (messages_received, bytes_received)

The OBS WebSocket protocol uses these opcodes:

- HELLO (0): Server greeting with auth info - handled by handle_hello_message
- IDENTIFY (1): Client auth - we send this, don't receive it
- IDENTIFIED (2): Auth success - handled by handle_identified_message
- EVENT (5): Real-time notifications - handled by handle_event_message
- REQUEST_RESPONSE (7): Command responses - handled by handle_request_response_message
- Other opcodes like REIDENTIFY, batch operations: not currently handled

This is one of the most critical functions in the library because it's in the hot path of message processing. Performance matters here. We keep it lightweight and defer heavy processing to the specific handlers.

Error handling is conservative: malformed JSON or missing opcode doesn't crash the connection, it just logs and continues. This allows us to be resilient to protocol variations or corruption.

Parameters

conn	The connection that received the message
message	Pointer to the raw message data (not null-terminated)
len	Number of bytes in the message

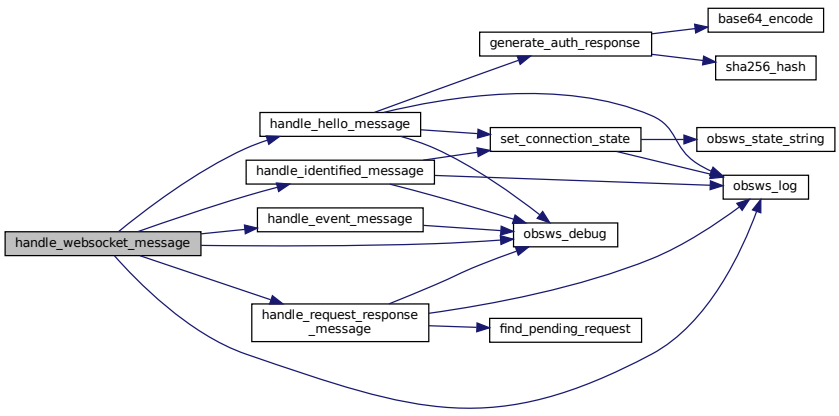
Returns

0 on success, -1 on parse error (does not disconnect)

Definition at line 932 of file library.c.

```
00932                                                                                                                {
00933     /* DEBUG_HIGH: Show full message content */
00934     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Received message (%zu bytes):  %.*s", len, (int)len,
      message);
00935
00936     cJSON *json = cJSON_ParseWithLength(message, len);
00937     if (!json) {
00938         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to parse JSON message");
00939         return -1;
00940     }
00941
00942     cJSON *op = cJSON_GetObjectItem(json, "op");
00943     cJSON *data = cJSON_GetObjectItem(json, "d");
00944
00945     if (!op) {
00946         obsws_log(conn, OBSWS_LOG_ERROR, "Message missing 'op' field");
00947         cJSON_Delete(json);
00948         return -1;
00949     }
00950
00951     /* DEBUG_MEDIUM: Show opcode being processed */
00952     obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Processing opcode:  %d", op->valueint);
00953
00954     int result = 0;
00955     switch (op->valueint) {
00956         case OBSWS_OPCODE_HELLO:
00957             result = handle_hello_message(conn, data);
00958             break;
00959         case OBSWS_OPCODE_IDENTIFIED:
00960             result = handle_identified_message(conn, data);
00961             break;
00962         case OBSWS_OPCODE_EVENT:
00963             result = handle_event_message(conn, data);
00964             break;
00965         case OBSWS_OPCODE_REQUEST_RESPONSE:
00966             result = handle_request_response_message(conn, data);
00967             break;
00968         default:
00969             obsws_log(conn, OBSWS_LOG_DEBUG, "Unhandled opcode:  %d", op->valueint);
00970             break;
00971     }
00972
00973     cJSON_Delete(json);
00974
00975     pthread_mutex_lock(&conn->stats_mutex);
00976     conn->stats.messages_received++;
00977     conn->stats.bytes_received += len;
00978     pthread_mutex_unlock(&conn->stats_mutex);
00979
00980     return result;
00981 }
```

Here is the call graph for this function:



Here is the caller graph for this function:

4.1.3.13 lws_callback()

```
static int lws_callback (
             struct lws * wsi,
             enum lws_callback_reasons reason,
             void * user,
             void * in,
             size_t len )  [static]
```

libwebsockets callback - routes WebSocket events to our handlers.

libwebsockets is an event-driven WebSocket library. It calls this callback function whenever something happens (connection established, data received, connection closed, etc.). The "reason" parameter identifies what happened.

We handle these key reasons:

- LWS_CALLBACK_CLIENT_ESTABLISHED: TCP/WebSocket handshake complete, ready for messages
- LWS_CALLBACK_CLIENT_RECEIVE: Data arrived from OBS
- LWS_CALLBACK_CLIENT_WRITEABLE: Socket is writable (less common with our design)
- LWS_CALLBACK_CLIENT_CONNECTION_ERROR: Connection failed (network error, bad host, etc.)
- LWS_CALLBACK_CLIENT_CLOSED: Connection closed normally
- LWS_CALLBACK_WSI_DESTROY: Cleanup callback

Important: This callback is called from the background event_thread, not the main application thread. So it must be thread-safe and not block.

Message assembly: OBS WebSocket messages might arrive fragmented (multiple packets). We accumulate them in recv_buffer and check lws_is_final_fragment() to know when a complete message has arrived. Only then do we parse it.

Error handling: Connection errors and receive buffer overflows are logged but don't crash. We just transition to ERROR state and let the connection cleanup/reconnection logic handle recovery.

Parameters

wsi	The WebSocket instance (provided by libwebsockets)
reason	The callback reason (LWS_CALLBACK_∗)
user	Our connection pointer (registered at context creation)
in	Incoming data (for LWS_CALLBACK_CLIENT_RECEIVE)
len	Size of incoming data

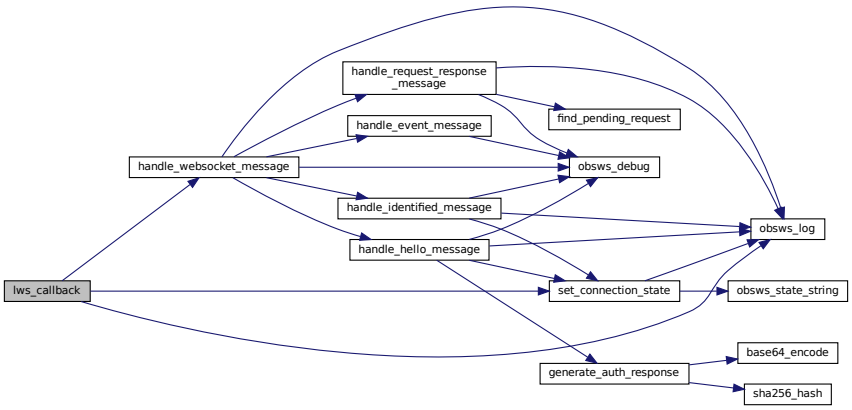
Returns

0 for success, -1 for error (affects connection handling)

Definition at line 1022 of file library.c.

```
01023                                                   {
01024     obsws_connection_t *conn = (obsws_connection_t *)user;
01025 
01026     switch (reason) {
01027         case LWS_CALLBACK_CLIENT_ESTABLISHED:
01028             obsws_log(conn, OBSWS_LOG_INFO, "WebSocket connection established");
01029             set_connection_state(conn, OBSWS_STATE_CONNECTING);
01030             break;
01031 
01032         case LWS_CALLBACK_CLIENT_RECEIVE:
01033             if (conn->recv_buffer_used + len < conn->recv_buffer_size) {
01034                 memcpy(conn->recv_buffer + conn->recv_buffer_used, in, len);
01035                 conn->recv_buffer_used += len;
01036 
01037                 if (lws_is_final_fragment(wsi)) {
01038                     handle_websocket_message(conn, conn->recv_buffer, conn->recv_buffer_used);
01039                     conn->recv_buffer_used = 0;
01040                 }
01041             } else {
01042                 obsws_log(conn, OBSWS_LOG_ERROR, "Receive buffer overflow");
01043                 conn->recv_buffer_used = 0;
01044             }
01045             break;
01046 
01047         case LWS_CALLBACK_CLIENT_WRITEABLE:
01048             /* Handle queued sends if needed */
01049             break;
01050 
01051         case LWS_CALLBACK_CLIENT_CONNECTION_ERROR:
01052             obsws_log(conn, OBSWS_LOG_ERROR, "Connection error: %s", in ? (char *)in : "unknown");
01053             set_connection_state(conn, OBSWS_STATE_ERROR);
01054             break;
01055 
01056         case LWS_CALLBACK_CLIENT_CLOSED:
01057             obsws_log(conn, OBSWS_LOG_INFO, "WebSocket connection closed (reason in 'in' param)");
01058             if (in && len > 0) {
01059                 obsws_log(conn, OBSWS_LOG_INFO, "Close reason: %.*s", (int)len, (char*)in);
01060             }
01061             set_connection_state(conn, OBSWS_STATE_DISCONNECTED);
01062             break;
01063 
01064         case LWS_CALLBACK_WSI_DESTROY:
01065             conn->wsi = NULL;
01066             break;
01067 
01068         default:
01069             break;
01070     }
01071 
01072     return 0;
01073 }
```

Here is the call graph for this function:



4.1.3.14 obsws_cleanup()

```
void obsws_cleanup (
            void  )
```

Clean up library resources.

Cleanup the OBS WebSocket library.

Call this when you're done with the library to deallocate OpenSSL resources. This is a counterpart to obsws_init().

Important: Make sure all obsws_connection_t objects have been disconnected and freed via obsws_disconnect() before calling this. If not, you might have dangling references and resource leaks.

Thread safety: This function is thread-safe and idempotent (safe to call multiple times). It checks g_library_initialized before doing anything.

Note: This is optional on program exit because the OS will clean up memory anyway. But it's good practice for:

- Library consumers that need clean shutdown
- Memory leak detectors / Valgrind tests
- Programs that unload the library

See also

obsws_init

Definition at line 1237 of file library.c.

```
01237                                {
01238     pthread_mutex_lock(&g_init_mutex);
01239
01240     if (!g_library_initialized) {
01241         pthread_mutex_unlock(&g_init_mutex);
01242         return;
01243     }
01244
01245     EVP_cleanup();
01246     g_library_initialized = false;
01247
01248     pthread_mutex_unlock(&g_init_mutex);
01249 }
```

4.1.3.15 obsws_config_init()

```
void obsws_config_init (
            obsws_config_t * config )
```

Initialize a connection configuration structure with safe defaults.

Create a default configuration structure with reasonable defaults.

Before calling obsws_connect(), you create an obsws_config_t structure with the connection parameters. This function initializes that structure with sensible defaults so you only need to change what's different for your use case.

Default values set:

- port: 4455 (OBS WebSocket v5 default port)
- use_ssl: false (OBS uses ws://, not wss://)
- connect_timeout_ms: 5000 (5 seconds to connect)
- recv_timeout_ms: 5000 (5 seconds to receive each message)
- send_timeout_ms: 5000 (5 seconds to send each message)
- ping_interval_ms: 10000 (send ping every 10 seconds)
- ping_timeout_ms: 5000 (expect pong within 5 seconds)
- auto_reconnect: true (reconnect automatically if connection drops)
- reconnect_delay_ms: 1000 (start with 1 second delay)
- max_reconnect_delay_ms: 30000 (max wait is 30 seconds)
- max_reconnect_attempts: 0 (infinite attempts)

After calling this, you typically set:

- config.host = "localhost" (where OBS is running)
- config.password = "your_password" (if OBS has auth enabled)
- config.event_callback = your_callback_func (to receive events)

Parameters

config	Pointer to structure to initialize (must not be NULL)

See also

obsws_connect

Definition at line 1367 of file library.c.

```
01367                                                {
01368     memset(config, 0, sizeof(obsws_config_t));
01369
01370     config->port = 4455;
01371     config->use_ssl = false;
01372     config->connect_timeout_ms = 5000;
01373     config->recv_timeout_ms = 5000;
01374     config->send_timeout_ms = 5000;
01375     config->ping_interval_ms = 10000;
01376     config->ping_timeout_ms = 5000;
01377     config->auto_reconnect = true;
01378     config->reconnect_delay_ms = 1000;
01379     config->max_reconnect_delay_ms = 30000;
01380     config->max_reconnect_attempts = 0; /* Infinite */
01381 }
```

4.1.3.16 obsws_connect()

```
obsws_connection_t * obsws_connect (
            const obsws_config_t * config )
```

Establish a connection to OBS.

This is the main entry point for using the library. You provide a configuration structure (initialized with obsws_config↩ _init and then customized), and this function connects to OBS, authenticates if needed, and spawns a background thread to handle incoming messages and events.

The function returns immediately - it doesn't wait for the connection to complete. Instead, it:

1. Creates a connection structure with the provided config
2. Allocates buffers for sending and receiving messages
3. Creates a libwebsockets context and connects to OBS
4. Spawns a background event_thread to process WebSocket messages
5. Returns the connection handle

Connection states: The connection progresses through states:

- DISCONNECTED -> CONNECTING (TCP handshake, WebSocket upgrade)
- CONNECTING -> AUTHENTICATING (receive HELLO, send IDENTIFY)
- AUTHENTICATING -> CONNECTED (receive IDENTIFIED)

You don't have to wait for CONNECTED state before calling obsws_send_request, but requests sent while not connected will return OBSWS_ERROR_NOT_CONNECTED.

Memory ownership: The connection structure is allocated and owned by this function. You must free it by calling obsws_disconnect(). Don't free it directly with free() - that will cause memory leaks (threads won't be cleaned up properly).

Error cases:

- NULL config or config->host: Returns NULL
- libwebsockets context creation fails: Returns NULL and logs error
- Network connection fails: Returns valid pointer but connection stays in ERROR state
- Bad password: Returns valid pointer but stays in AUTHENTICATING (never reaches CONNECTED)

Note: This function calls obsws_init() automatically if the library isn't already initialized.

Parameters

config	Pointer to initialized obsws_config_t with connection parameters

Returns

Pointer to new connection handle, or NULL if creation failed

See also

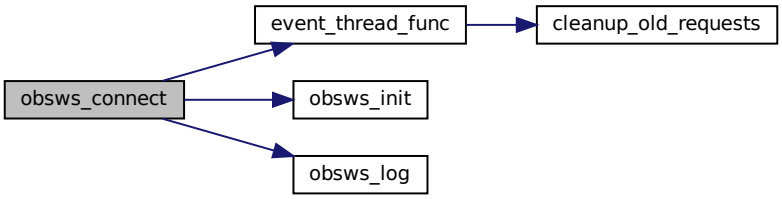
obsws_disconnect, obsws_get_state, obsws_send_request

Definition at line 1425 of file library.c.

```
01425                                                                   {
01426     if (!g_library_initialized) {
01427         obsws_init();
01428     }
01429
01430     if (!config || !config->host) {
01431         return NULL;
01432     }
01433
01434     obsws_connection_t *conn = calloc(1, sizeof(obsws_connection_t));
01435     if (!conn) return NULL;
01436
01437     /* Copy configuration */
01438     memcpy(&conn->config, config, sizeof(obsws_config_t));
01439     if (config->host) conn->config.host = strdup(config->host);
01440     if (config->password) conn->config.password = strdup(config->password);
01441
01442     /* Initialize mutexes */
01443     pthread_mutex_init(&conn->state_mutex, NULL);
01444     pthread_mutex_init(&conn->send_mutex, NULL);
01445     pthread_mutex_init(&conn->requests_mutex, NULL);
01446     pthread_mutex_init(&conn->stats_mutex, NULL);
01447     pthread_mutex_init(&conn->scene_mutex, NULL);
01448
01449     /* Allocate buffers */
01450     conn->recv_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01451     conn->recv_buffer = malloc(conn->recv_buffer_size);
01452     conn->send_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01453     conn->send_buffer = malloc(conn->send_buffer_size);
01454
01455     conn->state = OBSWS_STATE_DISCONNECTED;
01456     conn->current_reconnect_delay = config->reconnect_delay_ms;
01457
01458     /* Create libwebsockets context */
01459     struct lws_context_creation_info info;
01460     memset(&info, 0, sizeof(info));
01461
01462     info.port = CONTEXT_PORT_NO_LISTEN;
01463     info.protocols = protocols;
01464     info.gid = -1;
01465     info.uid = -1;
01466     info.options = LWS_SERVER_OPTION_DO_SSL_GLOBAL_INIT;
01467
01468     conn->lws_context = lws_create_context(&info);
01469     if (!conn->lws_context) {
01470         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to create libwebsockets context");
01471         free(conn->recv_buffer);
01472         free(conn->send_buffer);
01473         free(conn);
01474         return NULL;
01475     }
01476
01477     /* Connect to OBS */
01478     struct lws_client_connect_info ccinfo;
01479     memset(&ccinfo, 0, sizeof(ccinfo));
01480
01481     ccinfo.context = conn->lws_context;
01482     ccinfo.address = conn->config.host;
01483     ccinfo.port = conn->config.port;
01484     ccinfo.path = "/";
01485     ccinfo.host = ccinfo.address;
01486     ccinfo.origin = ccinfo.address;
01487     ccinfo.protocol = protocols[0].name;
01488     ccinfo.userdata = conn;
01489
01490     if (config->use_ssl) {
01491         ccinfo.ssl_connection = LCCSCF_USE_SSL;
01492     }
01493
01494     conn->wsi = lws_client_connect_via_info(&ccinfo);
01495     if (!conn->wsi) {
01496         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to initiate connection");
01497         lws_context_destroy(conn->lws_context);
01498         free(conn->recv_buffer);
01499         free(conn->send_buffer);
01500         free(conn);
01501         return NULL;
01502     }
01503
01504     /* Start event thread - protect flags with mutex */
```

```
01505     pthread_mutex_lock(&conn->state_mutex);
01506     conn->thread_running = true;
01507     conn->should_exit = false;
01508     pthread_mutex_unlock(&conn->state_mutex);
01509
01510     pthread_create(&conn->event_thread, NULL, event_thread_func, conn);
01511
01512     obsws_log(conn, OBSWS_LOG_INFO, "Connecting to OBS at %s:%d", config->host, config->port);
01513
01514     return conn;
01515 }
```

Here is the call graph for this function:



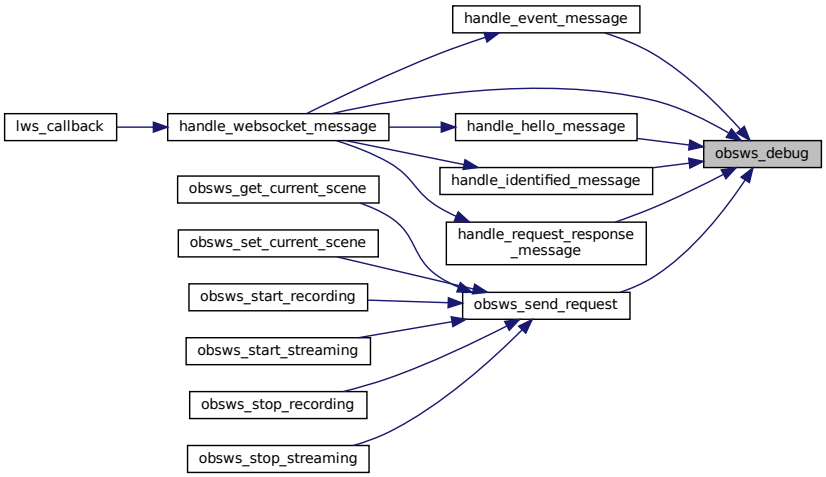
4.1.3.17 obsws_debug()

```
static void obsws_debug (
            obsws_connection_t * conn,
            obsws_debug_level_t min_level,
            const char * format,
             ... ) [static]
```

Definition at line 302 of file library.c.

```
00302
    {
00303     /* Only output if global debug level is at or above the minimum for this message */
00304     if (g_debug_level < min_level) {
00305         return;
00306     }
00307
00308     /* Format with a larger buffer for JSON and other verbose output */
00309     char message[4096];
00310     va_list args;
00311     va_start(args, format);
00312     vsnprintf(message, sizeof(message), format, args);
00313     va_end(args);
00314
00315     /* Route through the callback as DEBUG-level logs */
00316     if (conn && conn->config.log_callback) {
00317         conn->config.log_callback(OBSWS_LOG_DEBUG, message, conn->config.user_data);
00318     } else {
00319         const char *debug_level_str[] = {"NONE", "LOW", "MED", "HIGH"};
00320         fprintf(stderr, "[DEBUG-%s] %s\n", debug_level_str[min_level], message);
00321     }
00322 }
```

Here is the caller graph for this function:



4.1.3.18 obsws_disconnect()

```
void obsws_disconnect (
            obsws_connection_t * conn )
```

Disconnect from OBS and clean up connection resources.

Disconnect and destroy a connection.

This is the counterpart to obsws_connect(). It cleanly shuts down the connection, stops the background event thread, and frees all allocated resources.

The function performs these steps:

1. Signal the event_thread to stop by setting should_exit flag
2. Wait for the event_thread to actually exit using pthread_join()
3. Send a normal WebSocket close frame to OBS (if connected)
4. Destroy the libwebsockets context
5. Free all pending requests (they won't get responses now, but don't leak memory)
6. Free buffers, config, authentication data
7. Destroy all mutexes and condition variables
8. Free the connection structure

After calling this, the connection pointer is invalid. Don't use it again.

Safe to call multiple times: If you call it twice, the second call will be a no-op (because conn will be NULL).

Safe to call even if connection never fully established: If you disconnect while in CONNECTING or AUTHENTICATING state, everything is still cleaned up.

Important: This function blocks until the event_thread exits. If you have a callback that's blocked, this will deadlock. Make sure your callbacks don't block!

Parameters

conn	The connection to close (can be NULL - safe to call)

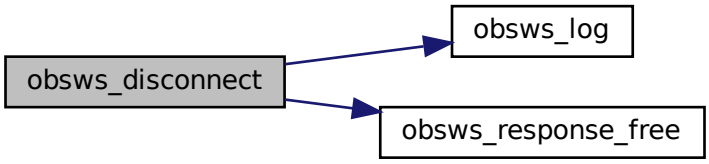
See also

obsws_connect

Definition at line 1549 of file library.c.

```
01549                                                {
01550     if (!conn) return;
01551
01552     obsws_log(conn, OBSWS_LOG_INFO, "Disconnecting from OBS");
01553
01554     /* Stop event thread - protect flag with mutex */
01555     pthread_mutex_lock(&conn->state_mutex);
01556     conn->should_exit = true;
01557     bool thread_was_running = conn->thread_running;
01558     pthread_mutex_unlock(&conn->state_mutex);
01559
01560     if (thread_was_running) {
01561         pthread_join(conn->event_thread, NULL);
01562     }
01563
01564     /* Close WebSocket - only if connected */
01565     if (conn->wsi && conn->state == OBSWS_STATE_CONNECTED) {
01566         lws_close_reason(conn->wsi, LWS_CLOSE_STATUS_NORMAL, NULL, 0);
01567     }
01568
01569     /* Cleanup libwebsockets */
01570     if (conn->lws_context) {
01571         lws_context_destroy(conn->lws_context);
01572     }
01573
01574     /* Free pending requests */
01575     pthread_mutex_lock(&conn->requests_mutex);
01576     pending_request_t *req = conn->pending_requests;
01577     while (req) {
01578         pending_request_t *next = req->next;
01579         if (req->response) {
01580             obsws_response_free(req->response);
01581         }
01582         pthread_mutex_destroy(&req->mutex);
01583         pthread_cond_destroy(&req->cond);
01584         free(req);
01585         req = next;
01586     }
01587     pthread_mutex_unlock(&conn->requests_mutex);
01588
01589     /* Free resources */
01590     free(conn->recv_buffer);
01591     free(conn->send_buffer);
01592     free((char *)conn->config.host);
01593     free((char *)conn->config.password);
01594     free(conn->challenge);
01595     free(conn->salt);
01596     free(conn->current_scene);
01597
01598     /* Destroy mutexes */
01599     pthread_mutex_destroy(&conn->state_mutex);
01600     pthread_mutex_destroy(&conn->send_mutex);
01601     pthread_mutex_destroy(&conn->requests_mutex);
01602     pthread_mutex_destroy(&conn->stats_mutex);
01603     pthread_mutex_destroy(&conn->scene_mutex);
01604
01605     free(conn);
01606 }
```

Here is the call graph for this function:



4.1.3.19 obsws_error_string()

```
const char * obsws_error_string (
            obsws_error_t error )
```

Convert an error code to a human-readable string.

Utility function for error reporting and logging. Returns a brief English description of each error code.

Never returns NULL Unknown error codes return "Unknown error", so it's always safe to use the returned pointer without NULL checks.

Example usage:

```
obsws_error_t err = obsws_send_request(...);
if (err != OBSWS_OK) {
    fprintf(stderr, "Error:  %s\\n", obsws_error_string(err));
}
```

Strings are constants Returned strings are statically allocated - do not modify or free them.

Parameters

error	The error code to describe

Returns

Pointer to a static string describing the error

See also

obsws_error_t

Definition at line 2229 of file library.c.

```
02229                                                {
02230     switch (error) {
02231         case OBSWS_OK:  return "Success";
02232         case OBSWS_ERROR_INVALID_PARAM:  return "Invalid parameter";
02233         case OBSWS_ERROR_CONNECTION_FAILED:  return "Connection failed";
```

```
02234         case OBSWS_ERROR_AUTH_FAILED:  return "Authentication failed";
02235         case OBSWS_ERROR_TIMEOUT:  return "Timeout";
02236         case OBSWS_ERROR_SEND_FAILED:  return "Send failed";
02237         case OBSWS_ERROR_RECV_FAILED:  return "Receive failed";
02238         case OBSWS_ERROR_PARSE_FAILED:  return "Parse failed";
02239         case OBSWS_ERROR_NOT_CONNECTED:  return "Not connected";
02240         case OBSWS_ERROR_ALREADY_CONNECTED:  return "Already connected";
02241         case OBSWS_ERROR_OUT_OF_MEMORY:  return "Out of memory";
02242         case OBSWS_ERROR_SSL_FAILED:  return "SSL failed";
02243         default:  return "Unknown error";
02244     }
02245 }
```

4.1.3.20 obsws_get_current_scene()

```
obsws_error_t obsws_get_current_scene (
            obsws_connection_t * conn,
            char * scene_name,
            size_t buffer_size )
```

Query the currently active scene in OBS.

This function queries OBS for the active scene name and returns it in the provided buffer. It also updates the local scene cache to keep it synchronized.

Cache Synchronization This function always queries the OBS server (doesn't use cached value). When the response arrives, it updates the cache. This ensures the library's cached scene name stays in sync with the actual OBS state.

Buffer Management The caller provides a buffer. If the scene name is longer than buffer_size-1, it will be truncated and null-terminated. Always check the returned buffer length if you need the full name.

Thread-safety The scene_mutex protects the cache update, so this is safe to call from any thread. Multiple concurrent calls are safe but will all query OBS (no deduplication).

Example usage:

```
char scene[256];
if (obsws_get_current_scene(conn, scene, sizeof(scene)) == OBSWS_OK) {
    printf("Current scene:  %s\\n", scene);
}
```

Parameters

conn	Connection object (must be in CONNECTED state)
scene_name	Buffer to receive the scene name (must not be NULL)
buffer_size	Size of scene_name buffer (must be $>$ 0)

Returns

OBSWS_OK if scene name retrieved successfully

OBSWS_ERROR_INVALID_PARAM if conn, scene_name is NULL or buffer_size is 0

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

OBSWS_ERROR_PARSE_FAILED if response can't be parsed

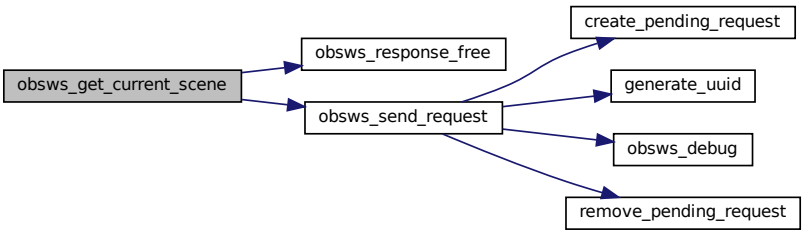
See also

obsws_set_current_scene

Definition at line 2022 of file library.c.

```
02022
     {
02023     if (!conn || !scene_name || buffer_size == 0) {
02024         return OBSWS_ERROR_INVALID_PARAM;
02025     }
02026
02027     obsws_response_t *response = NULL;
02028     obsws_error_t result = obsws_send_request(conn, "GetCurrentProgramScene", NULL, &response, 0);
02029
02030     if (result == OBSWS_OK && response && response->success && response->response_data) {
02031         cJSON *data = cJSON_Parse(response->response_data);
02032         if (data) {
02033             cJSON *name = cJSON_GetObjectItem(data, "currentProgramSceneName");
02034             if (name && name->valuestring) {
02035                 strncpy(scene_name, name->valuestring, buffer_size - 1);
02036                 scene_name[buffer_size - 1] = '\0';
02037
02038                 /* Update cache */
02039                 pthread_mutex_lock(&conn->scene_mutex);
02040                 free(conn->current_scene);
02041                 conn->current_scene = strdup(name->valuestring);
02042                 pthread_mutex_unlock(&conn->scene_mutex);
02043             }
02044             cJSON_Delete(data);
02045         }
02046     }
02047
02048     if (response) {
02049         obsws_response_free(response);
02050     }
02051
02052     return result;
02053 }
```

Here is the call graph for this function:



4.1.3.21 obsws_get_debug_level()

```
obsws_debug_level_t obsws_get_debug_level (
            void  )
```

Get the current debug level.

Get the current global debug level.

This is a read-only query - it doesn't change anything, just returns the current global debug level that was set by obsws_set_debug_level().

Useful for conditional logging in your application, e.g.:

```
if (obsws_get_debug_level() >= OBSWS_DEBUG_MEDIUM) {
    // do expensive trace operation
}
```

Returns

The currently active debug level

See also

obsws_set_debug_level

Definition at line 1333 of file library.c.

```
01333                                           {
01334     return g_debug_level;
01335 }
```

4.1.3.22 obsws_get_state()

```
obsws_state_t obsws_get_state (
            const obsws_connection_t * conn )
```

Get the current connection state.

Get the detailed current connection state.

Returns one of the connection states:

- OBSWS_STATE_DISCONNECTED: Not connected, idle
- OBSWS_STATE_CONNECTING: TCP connection established, waiting for WebSocket handshake
- OBSWS_STATE_AUTHENTICATING: WebSocket established, waiting for auth response
- OBSWS_STATE_CONNECTED: Connected and ready for requests
- OBSWS_STATE_ERROR: Connection encountered an error

State transitions normally follow this flow: DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED

But with errors, you can also have: (any state) -> ERROR

You don't usually need to call this - just try to send requests and check the error code. But it's useful for monitoring and debugging.

Thread-safe: This function locks the state_mutex, so it's safe to call from any thread.

Parameters

conn	The connection to check (NULL is safe - returns DISCONNECTED)

Returns

The current connection state

See also

obsws_is_connected

Definition at line 1663 of file library.c.

```
01663                                                                   {
01664     if (!conn) return OBSWS_STATE_DISCONNECTED;
01665
01666     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
01667     obsws_state_t state = conn->state;
01668     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
01669
01670     return state;
01671 }
```

4.1.3.23 obsws_get_stats()

```
obsws_error_t obsws_get_stats (
            const obsws_connection_t * conn,
            obsws_stats_t * stats )
```

Retrieve performance and connectivity statistics.

Get connection statistics and performance metrics.

The library maintains statistics about the connection:

- messages_sent / messages_received: Count of WebSocket messages
- bytes_sent / bytes_received: Total bytes transmitted/received
- connected_since: Timestamp of when we reached CONNECTED state
- reconnect_count: How many times we've reconnected (0 if never disconnected)

These can be useful for:

- Monitoring connection health
- Detecting stalled connections (if bytes_received stops increasing)
- Debugging and performance profiling
- Health dashboards or logging

Thread-safe: This function acquires stats_mutex and copies the entire stats structure, so it's safe to call from any thread. The copy operation is atomic from the caller's perspective.

Example usage:

```
obsws_stats_t stats;
obsws_get_stats(conn, &stats);
printf("Received %zu messages, %zu bytes\\n",
       stats.messages_received, stats.bytes_received);
```

Parameters

conn	The connection to query (NULL returns error)
stats	Pointer to stats structure to fill (must not be NULL)

Returns

OBSWS_OK on success, OBSWS_ERROR_INVALID_PARAM if conn or stats is NULL

See also

obsws_stats_t

Definition at line 1706 of file library.c.

```
01706                                                                                {
01707     if (!conn || !stats) return OBSWS_ERROR_INVALID_PARAM;
01708
01709     pthread_mutex_lock((pthread_mutex_t *)&conn->stats_mutex);
01710     memcpy(stats, &conn->stats, sizeof(obsws_stats_t));
01711     pthread_mutex_unlock((pthread_mutex_t *)&conn->stats_mutex);
01712
01713     return OBSWS_OK;
01714 }
```

4.1.3.24 obsws_init()

```
obsws_error_t obsws_init (
            void  )
```

Initialize the libwsv5 library.

Initialize the OBS WebSocket library.

This function must be called before creating any connections. It:

1. Initializes OpenSSL (EVP library for hashing)
2. Seeds the random number generator for UUID generation
3. Sets the global g_library_initialized flag

Thread safety: This function is thread-safe. Multiple threads can call it simultaneously, and only one will actually do the initialization (protected by g_init_mutex). Subsequent calls are no-ops.

Note: obsws_connect() will call this automatically if you forget, so you don't *have* to call it explicitly. But doing so allows you to initialize in a controlled way, separate from connection creation.

Cleanup: When you're done with the library, call obsws_cleanup() to deallocate resources. This is technically optional on program exit (the OS cleans up anyway), but good practice for testing and library shutdown.

Returns

OBSWS_OK always (initialization cannot fail in the current design)

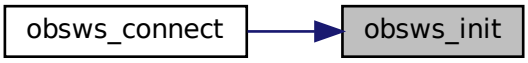
See also

obsws_cleanup, obsws_connect

Definition at line 1196 of file library.c.

```
01196                                {
01197     pthread_mutex_lock(&g_init_mutex);
01198 
01199     if (g_library_initialized) {
01200         pthread_mutex_unlock(&g_init_mutex);
01201         return OBSWS_OK;
01202     }
01203 
01204     /* Initialize OpenSSL */
01205     OpenSSL_add_all_algorithms();
01206 
01207     /* Seed random number generator */
01208     srand(time(NULL));
01209 
01210     g_library_initialized = true;
01211     pthread_mutex_unlock(&g_init_mutex);
01212 
01213     return OBSWS_OK;
01214 }
```

Here is the caller graph for this function:



4.1.3.25 obsws_is_connected()

```
bool obsws_is_connected (
            const obsws_connection_t * conn )
```

Check if a connection is actively connected to OBS.

Check if connection is currently authenticated and ready to use.

Convenience function that returns true if the connection is in CONNECTED state and false otherwise. Useful for checking before sending requests.

Thread-safe: This function locks the state_mutex before checking, so it's safe to call from any thread.

Return value: The connection must be in OBSWS_STATE_CONNECTED to return true. If it's CONNECTING, AUTHENTICATING, ERROR, or DISCONNECTED, this returns false.

Parameters

conn	The connection to check (NULL is safe - returns false)

Returns

true if connected, false otherwise

See also

obsws_get_state

Definition at line 1625 of file library.c.

```
01625                                                   {
01626     if (!conn) return false;
01627
01628     /* Thread-safe state check */
01629     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
01630     bool connected = (conn->state == OBSWS_STATE_CONNECTED);
01631     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
01632
01633     return connected;
01634 }
```

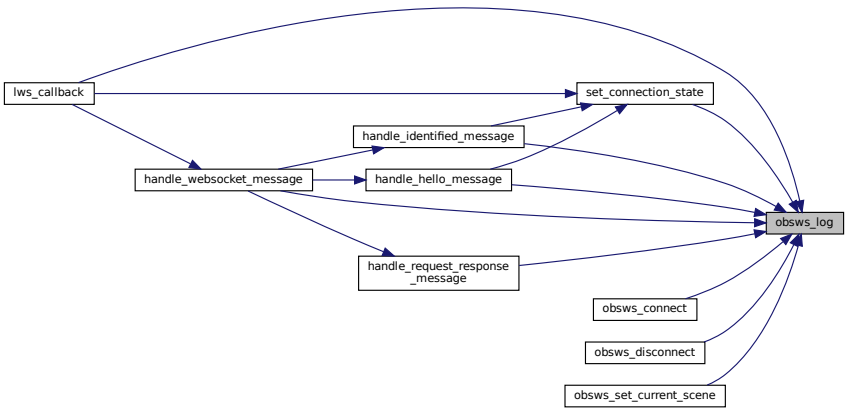
4.1.3.26 obsws_log()

```
static void obsws_log (
            obsws_connection_t * conn,
            obsws_log_level_t level,
            const char * format,
             ...  )  [static]
```

Definition at line 270 of file library.c.

```
00270                                                                                                   {
00271     /* Early exit if this message is too verbose */
00272     if (level > g_log_level) {
00273         return;
00274     }
00275
00276     /* Format the message using printf-style arguments */
00277     char message[1024];
00278     va_list args;
00279     va_start(args, format);
00280     vsnprintf(message, sizeof(message), format, args);
00281     va_end(args);
00282
00283     /* Route to user callback or stderr.  The callback lets the user handle logging
00284 however they want - write to file, send to logging service, etc.  */
00285     if (conn && conn->config.log_callback) {
00286         conn->config.log_callback(level, message, conn->config.user_data);
00287     } else {
00288         const char *level_str[] = {"NONE", "ERROR", "WARN", "INFO", "DEBUG"};
00289         fprintf(stderr, "[OBSWS-%s] %s\n", level_str[level], message);
00290     }
00291 }
```

Here is the caller graph for this function:



4.1.3.27 obsws_process_events()

```
int obsws_process_events (
            obsws_connection_t * conn,
            uint32_t timeout_ms )
```

Process pending WebSocket events (compatibility function).

Process pending events from the WebSocket connection.

This function is provided for API compatibility with single-threaded applications. However, the libwsv5 library uses a background event_thread by default, so this function is usually not needed - events are processed automatically in the background.

Background Event Processing: By design, all WebSocket messages (events, responses, etc.) are processed by the background event_thread. This thread:

- Continuously calls lws_service() to pump WebSocket events
- Receives incoming messages from OBS
- Routes responses to waiting callers via condition variables
- Calls event callbacks for real-time events
- Maintains keep-alive pings

Applications don't need to call this function - the thread handles everything.

What this function does: Currently, this is mainly for API compatibility. It:

- Validates the connection object
- If timeout_ms $>$ 0, sleeps for that duration
- Returns 0 (success)

When to use:

- Most applications: Don't call this - use background thread
- If you disable background thread: Call this in your main loop

Example (not typical - background thread is recommended):

```
// Not recommended - background thread is better
while (app_running) {
    obsws_process_events(conn, 100);  // Check every 100ms
}
```

Better approach - let background thread handle it:

```
obsws_connect(conn, "localhost", 4455, "password");
// Background thread processes events automatically
while (app_running) {
    // Your application code - no need to call process_events
}
obsws_disconnect(conn);
```

Parameters

conn	Connection object (can be NULL - returns error)
timeout_ms	Sleep duration in milliseconds (0 = don't sleep)

Returns

0 on success

OBSWS_ERROR_INVALID_PARAM if conn is NULL

See also

obsws_connect, obsws_disconnect, event_thread_func

Definition at line 2347 of file library.c.

```
02347                                                                   {
02348     if (!conn) return OBSWS_ERROR_INVALID_PARAM;
02349
02350     /* Events are processed in the background thread */
02351     /* This function is provided for API compatibility */
02352     if (timeout_ms > 0) {
02353         struct timespec ts;
02354         ts.tv_sec = timeout_ms / 1000;
02355         ts.tv_nsec = (timeout_ms % 1000) * 1000000;
02356         nanosleep(&ts, NULL);
02357     }
02358
02359     return 0;
02360 }
```

4.1.3.28 obsws_response_free()

```
void obsws_response_free (
            obsws_response_t * response )
```

Free a response object previously allocated by obsws_send_request().

Free a response structure and all its allocated memory.

This function safely deallocates all memory associated with a response:

- The error_message string (if present)
- The response_data JSON string (if present)
- The response structure itself

Safe to call with NULL Calling with NULL is safe and does nothing - no crash or error. This allows for simpler cleanup:

```
obsws_response_t *resp = NULL;
obsws_send_request(..., &resp, ...);
obsws_response_free(resp);  // Safe even if send_request failed
```

Memory ownership

- obsws_send_request() allocates the response - you must free it
- High-level functions like obsws_set_current_scene() can optionally take response ownership or free internally based on parameters

NOT thread-safe Each response should only be accessed/freed by one thread. If multiple threads need the response, use higher-level synchronization.

Parameters

response	Response to free. Can be NULL (does nothing if so).

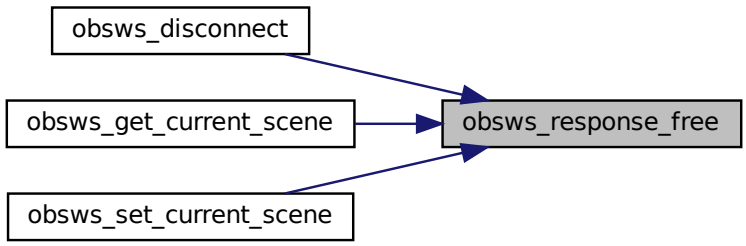
See also

obsws_send_request, obsws_response_t

Definition at line 2195 of file library.c.

```
02195                                                   {
02196     if (!response) return;
02197
02198     free(response->error_message);
02199     free(response->response_data);
02200     free(response);
02201 }
```

Here is the caller graph for this function:



4.1.3.29 obsws_send_request()

```
obsws_error_t obsws_send_request (
            obsws_connection_t * conn,
            const char * request_type,
            const char * request_data,
            obsws_response_t ** response,
            uint32_t timeout_ms )
```

Send a synchronous request to OBS and wait for the response.

This is the core function for all OBS operations. It implements the asynchronous request-response pattern of the OBS WebSocket v5 protocol:

Protocol Flow:

1. Generate a unique UUID for this request (used to match responses)
2. Create a pending_request_t to track the in-flight operation

3. Build the request JSON with opcode 6 (REQUEST)
4. Send the message via lws_write()
5. Block the caller with pthread_cond_timedwait() until response arrives
6. Return the response to caller (who owns it and must free with obsws_response_free)

Why synchronous from caller's perspective? Although WebSocket messages are async at the protocol level, we provide a synchronous API - the caller sends a request and blocks until the response arrives. This is simpler for application code than callback-based async APIs.

Behind the scenes, the background event_thread continuously processes WebSocket messages. When a REQUEST_RESPONSE (opcode 7) arrives matching a pending request ID, it signals the waiting condition variable, waking up the blocked caller.

Performance implications:

- Thread-safe: The main app thread can be blocked in obsws_send_request() while the background event_↩ thread processes other messages
- No polling: Uses condition variables, not CPU-wasting polling loops
- Can make multiple simultaneous requests from different threads (up to OBSWS_MAX_PENDING_↩ REQUESTS = 256)

Example usage:

```
obsws_response_t *response = NULL;
obsws_error_t err = obsws_send_request(conn, "SetCurrentProgramScene",
                                       "{\"sceneName\":  \"Scene1\"}",
                                       &response, 0);
if (err == OBSWS_OK && response && response->success) {
    printf("Scene switched successfully\\n");
}
obsws_response_free(response);
```

Parameters

conn	Connection object (must be in CONNECTED state)
request_type	OBS request type like "GetCurrentProgramScene", "SetCurrentProgramScene", etc.
request_data	Optional JSON string with request parameters. NULL for no parameters. Example: "{\"sceneName\": \"Scene1\"}"
response	Output pointer for the response. Will be allocated by this function. Caller must free with obsws_response_free(). Can be NULL if caller doesn't need the response (but response is still consumed from server).
timeout_ms	Timeout in milliseconds (0 = use config->recv_timeout_ms, typically 30000ms)

Returns

OBSWS_OK if response received (check response->success for operation success)

OBSWS_ERROR_INVALID_PARAM if conn, request_type, or response pointer is NULL

OBSWS_ERROR_NOT_CONNECTED if connection is not in CONNECTED state

OBSWS_ERROR_OUT_OF_MEMORY if pending request allocation fails

OBSWS_ERROR_SEND_FAILED if message send fails (buffer too small, invalid wsi, etc)

OBSWS_ERROR_TIMEOUT if no response received within timeout_ms

See also

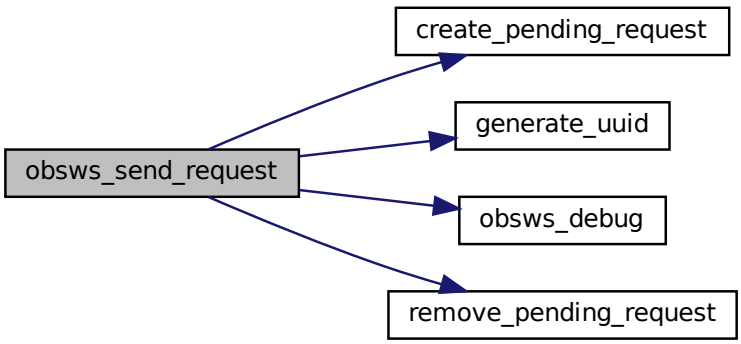
obsws_response_t, obsws_response_free, obsws_error_string

Definition at line 1776 of file library.c.

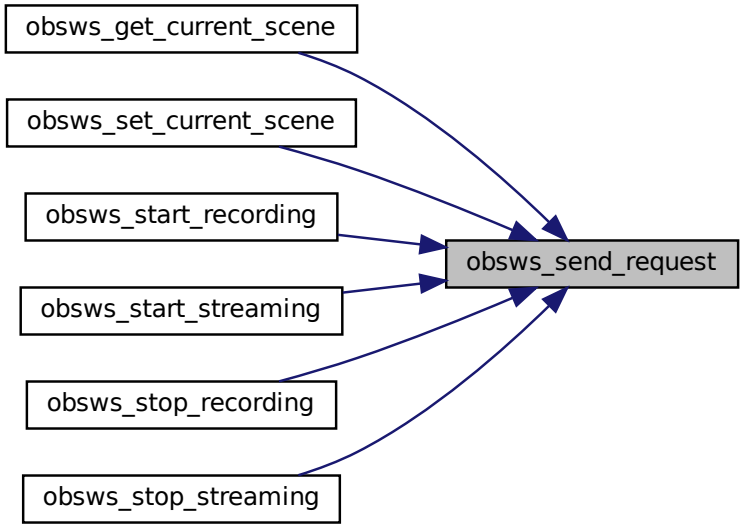
```
01777
      {
01778     if (!conn || !request_type || !response) {
01779         return OBSWS_ERROR_INVALID_PARAM;
01780     }
01781
01782     if (conn->state != OBSWS_STATE_CONNECTED) {
01783         return OBSWS_ERROR_NOT_CONNECTED;
01784     }
01785
01786     /* Generate request ID */
01787     char request_id[OBSWS_UUID_LENGTH];
01788     generate_uuid(request_id);
01789
01790     /* Create pending request */
01791     pending_request_t *req = create_pending_request(conn, request_id);
01792     if (!req) {
01793         return OBSWS_ERROR_OUT_OF_MEMORY;
01794     }
01795
01796     /* Build request JSON */
01797     cJSON *request = cJSON_CreateObject();
01798     cJSON_AddNumberToObject(request, "op", OBSWS_OPCODE_REQUEST);
01799
01800     cJSON *d = cJSON_CreateObject();
01801     cJSON_AddStringToObject(d, "requestType", request_type);
01802     cJSON_AddStringToObject(d, "requestId", request_id);
01803
01804     if (request_data) {
01805         cJSON *data = cJSON_Parse(request_data);
01806         if (data) {
01807             cJSON_AddItemToObject(d, "requestData", data);
01808         }
01809     }
01810
01811     cJSON_AddItemToObject(request, "d", d);
01812
01813     char *message = cJSON_PrintUnformatted(request);
01814     cJSON_Delete(request);
01815
01816     /* DEBUG_HIGH: Show request being sent */
01817     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending request (ID: %s):  %s", request_id, message);
01818
01819     /* Send request */
01820     pthread_mutex_lock(&conn->send_mutex);
01821     size_t len = strlen(message);
01822     obsws_error_t result = OBSWS_OK;
01823
01824     if (len < conn->send_buffer_size - LWS_PRE && conn->wsi) {
01825         memcpy(conn->send_buffer + LWS_PRE, message, len);
01826         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
      LWS_WRITE_TEXT);
01827
01828         if (written < 0) {
01829             result = OBSWS_ERROR_SEND_FAILED;
01830         } else {
01831             pthread_mutex_lock(&conn->stats_mutex);
01832             conn->stats.messages_sent++;
01833             conn->stats.bytes_sent += len;
01834             pthread_mutex_unlock(&conn->stats_mutex);
01835         }
01836     } else {
01837         result = OBSWS_ERROR_SEND_FAILED;
01838     }
01839     pthread_mutex_unlock(&conn->send_mutex);
01840
01841     free(message);
01842
01843     if (result != OBSWS_OK) {
01844         remove_pending_request(conn, req);
01845         return result;
01846     }
01847
01848     /* Wait for response */
01849     if (timeout_ms == 0) {
01850         timeout_ms = conn->config.recv_timeout_ms;
01851     }
01852
01853     struct timespec ts;
01854     clock_gettime(CLOCK_REALTIME, &ts);
```

```
01855     ts.tv_sec += timeout_ms / 1000;
01856     ts.tv_nsec += (timeout_ms % 1000) * 1000000;
01857     if (ts.tv_nsec >= 1000000000) {
01858         ts.tv_sec++;
01859         ts.tv_nsec -= 1000000000;
01860     }
01861
01862     pthread_mutex_lock(&req->mutex);
01863     while (!req->completed) {
01864         int wait_result = pthread_cond_timedwait(&req->cond, &req->mutex, &ts);
01865         if (wait_result == ETIMEDOUT) {
01866             pthread_mutex_unlock(&req->mutex);
01867             remove_pending_request(conn, req);
01868             return OBSWS_ERROR_TIMEOUT;
01869         }
01870     }
01871
01872     *response = req->response;
01873     req->response = NULL; /* Transfer ownership */
01874     pthread_mutex_unlock(&req->mutex);
01875
01876     remove_pending_request(conn, req);
01877
01878     return OBSWS_OK;
01879 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.1.3.30 obsws_set_current_scene()

```
obsws_error_t obsws_set_current_scene (
            obsws_connection_t * conn,
            const char * scene_name,
            obsws_response_t ** response )
```

Switch OBS to a specific scene.

This is a high-level convenience function for scene switching. It demonstrates several important design patterns in the library:

Optimization: Scene Cache Before sending a request to OBS, this function checks the cached current scene. If the requested scene is already active, it returns immediately without network overhead. The cache is maintained by the event thread when SceneChanged events arrive.

Memory Ownership If the caller provides response pointer, they receive ownership and must call obsws_response_free(). If response is NULL, the function frees the response internally. This flexibility allows three usage patterns:

1. Check response: `obsws_set_current_scene(conn, name, &resp); if (resp->success) ...`
2. Ignore response: `obsws_set_current_scene(conn, name, NULL);` (response is freed internally)
3. Just check error: `if (obsws_send_request(...) != OBSWS_OK) ...`

Example usage:

```
// Pattern 1:  Check response details
obsws_response_t *response = NULL;
if (obsws_set_current_scene(conn, "Scene1", &response) == OBSWS_OK &&
    response && response->success) {
    printf("Switched successfully\\n");
} else {
    printf("Switch failed:  %s\\n", response ?  response->error_message :  "unknown");
}
obsws_response_free(response);
// Pattern 2:  Ignore response (simpler)
obsws_set_current_scene(conn, "Scene1", NULL);
```

Thread-safety:

- Scene cache is protected by scene_mutex
- Safe to call from any thread
- Multiple calls can happen simultaneously (each uses send_mutex)

Parameters

conn	Connection object (must be in CONNECTED state)
scene_name	Name of the scene to switch to. Must not be NULL.
response	Optional output for response details. If provided, caller owns it and must free with obsws_response_free(). If NULL, response is freed internally.

Returns

OBSWS_OK if request sent and response received (check response->success for whether the scene switch actually succeeded in OBS)

OBSWS_ERROR_INVALID_PARAM if conn or scene_name is NULL

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if no response from OBS

See also

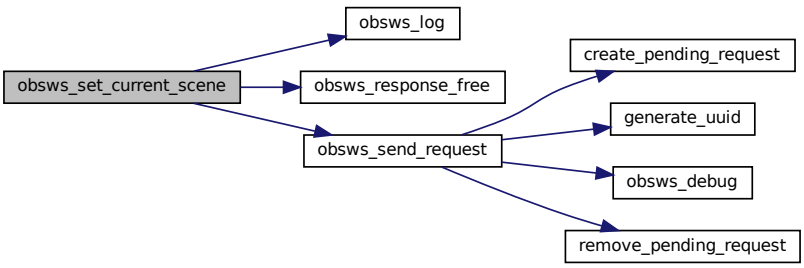
obsws_get_current_scene, obsws_response_t, obsws_response_free

Definition at line 1935 of file library.c.

```
01935
    {
01936     if (!conn || !scene_name) {
01937         return OBSWS_ERROR_INVALID_PARAM;
01938     }
01939
01940     /* Check cache to avoid redundant switches */
01941     pthread_mutex_lock(&conn->scene_mutex);
01942     bool already_current = (conn->current_scene && strcmp(conn->current_scene, scene_name) == 0);
01943     pthread_mutex_unlock(&conn->scene_mutex);
01944
01945     if (already_current) {
01946         obsws_log(conn, OBSWS_LOG_DEBUG, "Already on scene:  %s", scene_name);
01947         if (response) {
01948             *response = calloc(1, sizeof(obsws_response_t));
01949             (*response)->success = true;
01950         }
01951         return OBSWS_OK;
01952     }
01953
01954     cJSON *request_data = cJSON_CreateObject();
01955     cJSON_AddStringToObject(request_data, "sceneName", scene_name);
01956
01957     char *data_str = cJSON_PrintUnformatted(request_data);
```

```
01958     cJSON_Delete(request_data);
01959
01960     obsws_response_t *resp = NULL;
01961     obsws_error_t result = obsws_send_request(conn, "SetCurrentProgramScene", data_str, &resp, 0);
01962     free(data_str);
01963
01964     if (result == OBSWS_OK && resp && resp->success) {
01965         pthread_mutex_lock(&conn->scene_mutex);
01966         free(conn->current_scene);
01967         conn->current_scene = strdup(scene_name);
01968         pthread_mutex_unlock(&conn->scene_mutex);
01969
01970         obsws_log(conn, OBSWS_LOG_INFO, "Switched to scene:  %s", scene_name);
01971     }
01972
01973     if (response) {
01974         *response = resp;
01975     } else if (resp) {
01976         obsws_response_free(resp);
01977     }
01978
01979     return result;
01980 }
```

Here is the call graph for this function:



4.1.3.31 obsws_set_debug_level()

```
void obsws_set_debug_level (
            obsws_debug_level_t level )
```

Set the global debug level.

Set the global debug level for the library.

Debug logging is separate from regular logging. It provides extremely detailed trace information about the Web↩ Socket protocol, message parsing, authentication, etc. This is useful during development and troubleshooting.

Debug levels:

- OBSWS_DEBUG_NONE: No debug output (fastest)
- OBSWS_DEBUG_LOW: Major state transitions and connection events
- OBSWS_DEBUG_MEDIUM: Message types and handlers invoked
- OBSWS_DEBUG_HIGH: Full message content and every operation

Debug logging is independent of log level. You can have OBSWS_LOG_ERROR set (hide non-error logs) but still see OBSWS_DEBUG_HIGH output.

Performance warning: OBSWS_DEBUG_HIGH produces enormous output and will slow down the library significantly. Only use during debugging!

Thread safety: Same as obsws_set_log_level (modifies global without locking).

Parameters

level	The debug verbosity level

See also

obsws_set_log_level, obsws_get_debug_level

Definition at line 1313 of file library.c.

```
01313                                                   {
01314     g_debug_level = level;
01315 }
```

4.1.3.32 obsws_set_log_level()

```
void obsws_set_log_level (
            obsws_log_level_t level )
```

Set the global log level threshold.

Set the global log level for the library.

The library logs various messages during operation. This function sets which messages are displayed. All messages at the specified level and higher severity are shown; lower severity messages are hidden.

Levels in increasing severity:

- OBSWS_LOG_DEBUG: Low-level diagnostic info (too verbose for production)
- OBSWS_LOG_INFO: General informational messages (usual choice)
- OBSWS_LOG_WARNING: Potentially problematic situations (degraded but working)
- OBSWS_LOG_ERROR: Error conditions that need attention

Example: If you call obsws_set_log_level(OBSWS_LOG_WARNING), you'll see only WARNING and ERROR messages, but not INFO or DEBUG messages.

Thread safety: This modifies a global variable without locking, so if you might call this from multiple threads, use synchronization externally.

Parameters

level	The minimum severity level to display

See also

obsws_set_debug_level

Definition at line 1285 of file library.c.

```
01285                                                   {
01286     g_log_level = level;
01287 }
```

4.1.3.33 obsws_start_recording()

```
obsws_error_t obsws_start_recording (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Start recording in OBS.

Tells OBS to begin recording the current scene composition to disk. The recording path and format are determined by OBS settings, not by this library.

This is a convenience wrapper around obsws_send_request() using the OBS "StartRecord" request type.

Return value interpretation:

- OBSWS_OK: Request was sent and OBS responded (check response->success)
- Other errors: Network/connection problem

Example usage:

```
obsws_response_t *resp = NULL;
if (obsws_start_recording(conn, &resp) == OBSWS_OK && resp && resp->success) {
    printf("Recording started\\n");
}
obsws_response_free(resp);
```

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

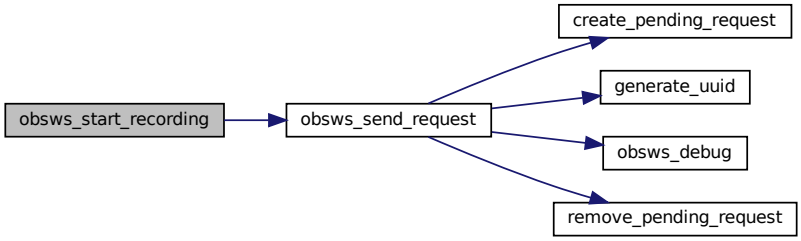
See also

obsws_stop_recording, obsws_send_request, obsws_response_free

Definition at line 2086 of file library.c.

```
02086                                                                                   {
02087     return obsws_send_request(conn, "StartRecord", NULL, response, 0);
02088 }
```

Here is the call graph for this function:



4.1.3.34 obsws_start_streaming()

```
obsws_error_t obsws_start_streaming (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Start streaming in OBS.

Tells OBS to begin streaming to the configured destination (Twitch, YouTube, etc). The stream settings (URL, key, bitrate, etc) are determined by OBS settings.

This is a convenience wrapper around obsws_send_request() using the OBS "StartStream" request type.

Thread-safe: Safe to call from any thread while connected.

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

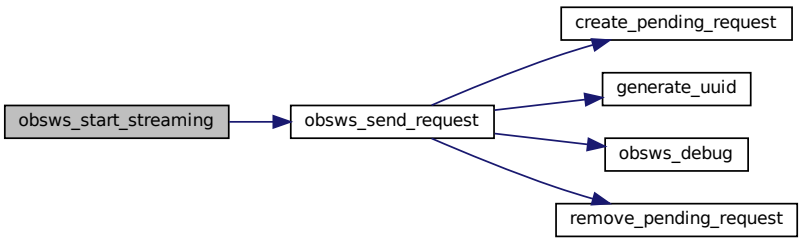
See also

obsws_stop_streaming, obsws_send_request, obsws_response_free

Definition at line 2139 of file library.c.

```
02139                                                                           {
02140     return obsws_send_request(conn, "StartStream", NULL, response, 0);
02141 }
```

Here is the call graph for this function:



4.1.3.35 obsws_state_string()

```
const char * obsws_state_string (
            obsws_state_t state )
```

Convert a connection state to a human-readable string.

Utility function for logging and debugging. Returns a brief English description of each connection state.

State Transitions:

- DISCONNECTED: Initial state or after disconnect()
- CONNECTING: connect() called, establishing TCP connection
- AUTHENTICATING: TCP connected, performing challenge-response auth
- CONNECTED: Auth succeeded, ready for requests
- ERROR: Network error or protocol failure, should reconnect

Valid transitions:

```
DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED
CONNECTED -> DISCONNECTED (on explicit disconnect)
CONNECTED -> ERROR (on network failure)
ERROR -> CONNECTING (on reconnect attempt)
```

Example usage:

```
obsws_state_t state = obsws_get_state(conn);
printf("Connection state:  %s\\n", obsws_state_string(state));
```

Never returns NULL - unknown states return "Unknown". Returned strings are static - do not modify or free.

Parameters

state	The connection state to describe

Returns

Pointer to a static string describing the state

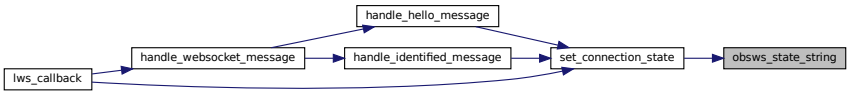
See also

obsws_get_state, obsws_state_t, obsws_is_connected

Definition at line 2282 of file library.c.

```
02282                                                 {
02283     switch (state) {
02284         case OBSWS_STATE_DISCONNECTED:  return "Disconnected";
02285         case OBSWS_STATE_CONNECTING:  return "Connecting";
02286         case OBSWS_STATE_AUTHENTICATING:  return "Authenticating";
02287         case OBSWS_STATE_CONNECTED:  return "Connected";
02288         case OBSWS_STATE_ERROR:  return "Error";
02289         default:  return "Unknown";
02290     }
02291 }
```

Here is the caller graph for this function:



4.1.3.36 obsws_stop_recording()

```
obsws_error_t obsws_stop_recording (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Stop recording in OBS.

Stop recording.

Tells OBS to stop the currently active recording. If no recording is in progress, OBS returns success anyway (idempotent operation).

This is a convenience wrapper around obsws_send_request() using the OBS "StopRecord" request type.

Example usage:

```
if (obsws_stop_recording(conn, NULL) != OBSWS_OK) {
    fprintf(stderr, "Failed to stop recording\\n");
}
```

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

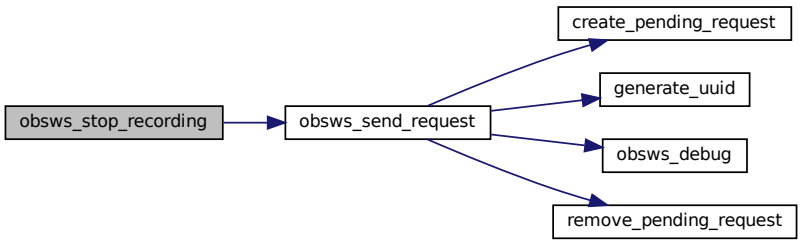
See also

obsws_start_recording, obsws_send_request, obsws_response_free

Definition at line 2115 of file library.c.

```
02115                                                                                {
02116     return obsws_send_request(conn, "StopRecord", NULL, response, 0);
02117 }
```

Here is the call graph for this function:



4.1.3.37 obsws_stop_streaming()

```
obsws_error_t obsws_stop_streaming (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Stop streaming in OBS.

Stop streaming.

Tells OBS to stop the active stream. If not currently streaming, OBS returns success anyway (idempotent operation).

This is a convenience wrapper around obsws_send_request() using the OBS "StopStream" request type.

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

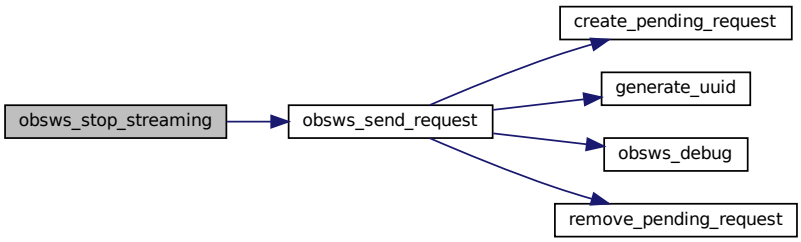
See also

obsws_start_streaming, obsws_send_request, obsws_response_free

Definition at line 2161 of file library.c.

```
02161                                                                                {
02162     return obsws_send_request(conn, "StopStream", NULL, response, 0);
02163 }
```

Here is the call graph for this function:



4.1.3.38 obsws_version()

```
const char * obsws_version (
            void  )
```

Get the library version string.

Returns a semantic version string like "1.0.0" that identifies which version of libwsv5 is being used. Useful for debugging and logging.

Returns

Pointer to static version string (don't free)

Definition at line 1259 of file library.c.

```
01259                                {
01260     return OBSWS_VERSION;
01261 }
```

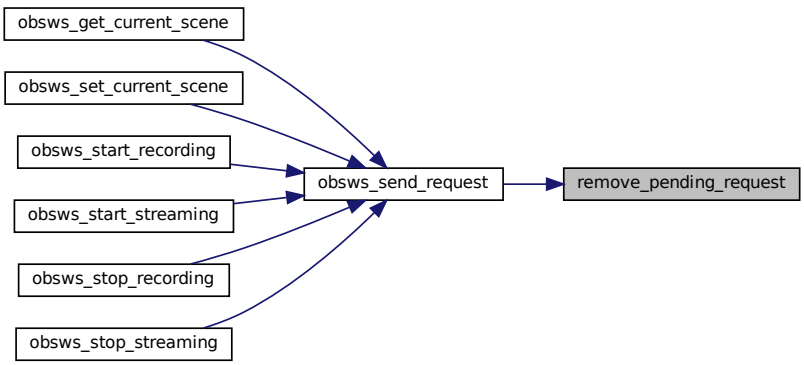
4.1.3.39 remove_pending_request()

```
static void remove_pending_request (
            obsws_connection_t * conn,
            pending_request_t * target )  [static]
```

Definition at line 549 of file library.c.

```
00549                                                                                {
00550     pthread_mutex_lock(&conn->requests_mutex);
00551     pending_request_t **req = &conn->pending_requests;
00552
00553     /* Find and remove from linked list */
00554     while (*req) {
00555         if (*req == target) {
00556             *req = target->next;
00557             pthread_mutex_unlock(&conn->requests_mutex);
00558
00559             /* Clean up request resources */
00560             pthread_mutex_destroy(&target->mutex);
00561             pthread_cond_destroy(&target->cond);
00562             free(target);
00563             return;
00564         }
00565         req = &(*req)->next;
00566     }
00567
00568     pthread_mutex_unlock(&conn->requests_mutex);
00569 }
```

Here is the caller graph for this function:



4.1.3.40 set_connection_state()

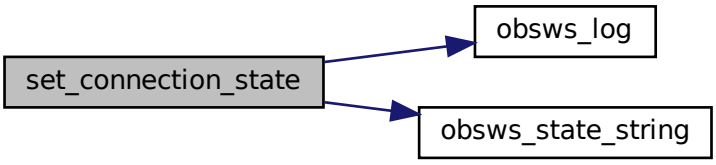
```
static void set_connection_state (
            obsws_connection_t * conn,
            obsws_state_t new_state )  [static]
```

Definition at line 477 of file library.c.

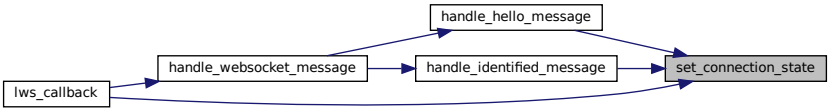
```
00477                                                                                {
00478     /* Acquire lock, save old state, set new state, release lock */
00479     pthread_mutex_lock(&conn->state_mutex);
00480     obsws_state_t old_state = conn->state;
00481     conn->state = new_state;
00482     pthread_mutex_unlock(&conn->state_mutex);
00483
```

```
00484     /* Call callback only if state actually changed (not a duplicate) */
00485     if (old_state != new_state && conn->config.state_callback) {
00486         conn->config.state_callback(conn, old_state, new_state, conn->config.user_data);
00487     }
00488
00489     /* Log the transition for debugging/monitoring */
00490     obsws_log(conn, OBSWS_LOG_INFO, "State changed:  %s -> %s",
00491               obsws_state_string(old_state), obsws_state_string(new_state));
00492 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



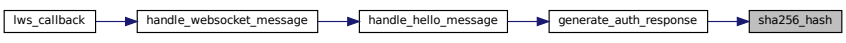
4.1.3.41 sha256_hash()

```
static void sha256_hash (
            const char * input,
            unsigned char * output )  [static]
```

Definition at line 411 of file library.c.

```
00411                                                                   {
00412     EVP_MD_CTX *ctx = EVP_MD_CTX_new();
00413     EVP_DigestInit_ex(ctx, EVP_sha256(), NULL);
00414     EVP_DigestUpdate(ctx, input, strlen(input));
00415     EVP_DigestFinal_ex(ctx, output, NULL);
00416     EVP_MD_CTX_free(ctx);
00417 }
```

Here is the caller graph for this function:



4.1.4 Variable Documentation

4.1.4.1 g_debug_level

```
obsws_debug_level_t g_debug_level = OBSWS_DEBUG_NONE  [static]
```

Definition at line 253 of file library.c.

4.1.4.2 g_init_mutex

```
pthread_mutex_t g_init_mutex = PTHREAD_MUTEX_INITIALIZER  [static]
```

Definition at line 254 of file library.c.

4.1.4.3 g_library_initialized

```
bool g_library_initialized = false  [static]
```

Definition at line 251 of file library.c.

4.1.4.4 g_log_level

```
obsws_log_level_t g_log_level = OBSWS_LOG_INFO  [static]
```

Definition at line 252 of file library.c.

4.1.4.5 protocols

```
const struct lws_protocols protocols[]  [static]
```

Initial value:

```
= {
    {
        "obs-websocket",
        lws_callback,
        0,
        OBSWS_DEFAULT_BUFFER_SIZE,
        0,
        NULL,
        0
    },
    { NULL, NULL, 0, 0, 0, NULL, 0 }
}
```

Definition at line 1075 of file library.c.

4.2 library.c

Go to the documentation of this file.

```
00001 #define _POSIX_C_SOURCE 200809L
00002
00003 #include "library.h"
00004
00005 #include <stdio.h>
00006 #include <stdlib.h>
00007 #include <string.h>
00008 #include <time.h>
00009 #include <pthread.h>
00010 #include <unistd.h>
00011 #include <stdarg.h>
00012 #include <sys/socket.h>
00013 #include <sys/select.h>
00014 #include <netinet/in.h>
00015 #include <arpa/inet.h>
00016 #include <netdb.h>
00017 #include <errno.h>
00018 #include <poll.h>
00019
00020 /* Third-party dependencies */
00021 #include <libwebsockets.h>
00022 #include <openssl/sha.h>
00023 #include <openssl/evp.h>
00024 #include <openssl/bio.h>
00025 #include <openssl/buffer.h>
00026 #include <cjson/cJSON.h>
00027
00028 /* ================================================================================
00029 * Constants and Macros
00030 * ================================================================================ */
00031
00032 #define OBSWS_VERSION "1.0.0"                  /* Library version string */
00033 #define OBSWS_PROTOCOL_VERSION 1               /* OBS WebSocket protocol version (v5 uses RPC version
      1) */
00034
00035 /* Buffer sizing:  64KB is large enough for most OBS messages.  Larger messages
00036 (like scene lists with many scenes) might need bigger buffers, but this is
00037 a reasonable default.  We could make it dynamic, but that adds complexity.
00038 The protocol itself doesn't define a max message size, so we have to choose.  */
00039 #define OBSWS_DEFAULT_BUFFER_SIZE 65536        /* 64KB buffer for WebSocket messages */
00040
00041 /* Pending requests tracking:  We use a linked list to track requests waiting for
00042 responses.  256 is a reasonable limit - you can have up to 256 requests in-flight
00043 at once.  In practice, most apps will have way fewer.  We chose a limit to prevent
00044 unbounded memory growth if something goes wrong and requests never complete.  */
00045 #define OBSWS_MAX_PENDING_REQUESTS 256
00046
00047 /* UUIDs are exactly 36 characters (8-4-4-4-12 hex digits with dashes) plus null terminator.
00048 We use these to match requests with their responses in the asynchronous protocol.  */
00049 #define OBSWS_UUID_LENGTH 37
00050
00051 /* OBS WebSocket v5 OpCodes - message type identifiers in the protocol.
00052
00053 The OBS WebSocket v5 protocol uses opcodes to identify message types.  The protocol
00054 is based on a request-response model layered on top of WebSocket.  Here's the flow:
00055
00056 1.  Server sends HELLO (opcode 0) with auth challenge and salt
00057 2.  Client sends IDENTIFY (opcode 1) with auth response and client info
00058 3.  Server sends IDENTIFIED (opcode 2) if auth succeeded
00059 4.  Client can send REQUEST messages (opcode 6)
00060 5.  Server responds with REQUEST_RESPONSE (opcode 7)
00061 6.  Server sends EVENT messages (opcode 5) for things happening in OBS
00062
00063 Batch operations (opcodes 8-9) let you send multiple requests in one message,
00064 but we don't use them in this library - each request is sent individually.
00065 REIDENTIFY (opcode 3) is used if we lose connection and reconnect.
00066 */
00067
00068 #define OBSWS_OPCODE_HELLO 0                   /* Server:  Initial greeting with auth info */
00069 #define OBSWS_OPCODE_IDENTIFY 1                /* Client:  Authentication and protocol agreement */
00070 #define OBSWS_OPCODE_IDENTIFIED 2              /* Server:  Auth successful, ready for commands */
00071 #define OBSWS_OPCODE_REIDENTIFY 3              /* Client:  Re-authenticate after reconnect */
00072 #define OBSWS_OPCODE_EVENT 5                   /* Server:  Something happened in OBS */
00073 #define OBSWS_OPCODE_REQUEST 6                 /* Client:  Execute an operation in OBS */
00074 #define OBSWS_OPCODE_REQUEST_RESPONSE 7        /* Server:  Result of a client request */
00075 #define OBSWS_OPCODE_REQUEST_BATCH 8           /* Client:  Multiple requests at once (unused) */
00076 #define OBSWS_OPCODE_REQUEST_BATCH_RESPONSE 9  /* Server:  Responses to batch (unused) */
00077
00078 /* Event subscription flags - bitmask for which OBS event categories we subscribe to.
00079
00080 The OBS WebSocket protocol lets you specify which events you want to receive.  This
00081 avoids bandwidth waste - if you don't care about media playback events, don't subscribe.
```

```
00082 We subscribe to most categories by default (using a bitmask), but you could modify
00083 this to be more selective if needed.
00084
00085 We chose a bitmask (0x7FF for all) rather than subscribing/unsubscribing individually
00086 because it's more efficient - one subscription message at connect-time instead of
00087 many individual subscribe/unsubscribe messages.
00088 */
00089
00090 #define OBSWS_EVENT_GENERAL (1 << 0)        /* General OBS events (startup, shutdown) */
00091 #define OBSWS_EVENT_CONFIG (1 << 1)         /* Configuration change events */
00092 #define OBSWS_EVENT_SCENES (1 << 2)         /* Scene-related events (scene switched, etc) */
00093 #define OBSWS_EVENT_INPUTS (1 << 3)         /* Input source events (muted, volume changed) */
00094 #define OBSWS_EVENT_TRANSITIONS (1 << 4)    /* Transition events (transition started) */
00095 #define OBSWS_EVENT_FILTERS (1 << 5)        /* Filter events (filter added, removed) */
00096 #define OBSWS_EVENT_OUTPUTS (1 << 6)        /* Output events (recording started, streaming stopped) */
00097 #define OBSWS_EVENT_SCENE_ITEMS (1 << 7)    /* Scene item events (source added to scene) */
00098 #define OBSWS_EVENT_MEDIA_INPUTS (1 << 8)   /* Media playback events (media finished) */
00099 #define OBSWS_EVENT_VENDORS (1 << 9)        /* Vendor-specific extensions */
00100 #define OBSWS_EVENT_UI (1 << 10)            /* UI events (Studio Mode toggled) */
00101 #define OBSWS_EVENT_ALL 0x7FF                /* Subscribe to all event types */
00102
00103 /* ========================================================================
00104 * Internal Structures
00105 * ======================================================================== */
00106
00107 /* Pending request tracking - manages asynchronous request/response pairs.
00108
00109 The OBS WebSocket protocol is asynchronous - when you send a request, you don't
00110 wait for the response before continuing.  Instead, responses come back later with
00111 a request ID matching them to the original request.
00112
00113 This struct tracks one in-flight request.  We keep a linked list of these, one for
00114 each request waiting for a response.  When a response arrives, we find the matching
00115 pending_request by ID, populate the response field, and set completed=true.  The
00116 thread that sent the request is waiting on the condition variable, so it wakes up
00117 and gets the response.
00118
00119 Why use a condition variable instead of polling?  Because polling wastes CPU. A
00120 thread waiting on a condition variable goes to sleep until the response arrives,
00121 at which point it's woken up.  Much more efficient.
00122
00123 Why use a timestamp?  For timeout detection.  If a response never arrives (OBS crashed,
00124 network died, etc.), we detect it by checking if the request is older than the timeout.
00125 */
00126
00127 typedef struct pending_request {
00128     char request_id[OBSWS_UUID_LENGTH];     /* Unique ID matching request to response */
00129     obsws_response_t *response;             /* Response data populated when received */
00130     bool completed;                         /* Flag indicating response received */
00131     pthread_mutex_t mutex;                  /* Protects the response/completed fields */
00132     pthread_cond_t cond;                    /* Waiting thread sleeps here until response arrives */
00133     time_t timestamp;                       /* When request was created - used for timeout detection
    */
00134     struct pending_request *next;           /* Linked list pointer to next pending request */
00135 } pending_request_t;
00136
00137 /* Main connection structure - holds all state for an OBS WebSocket connection.
00138
00139 This is the main opaque type that users interact with.  It holds everything needed
00140 to manage one connection to OBS:
00141 - Configuration (where to connect, timeouts, callbacks)
00142 - WebSocket instance (from libwebsockets)
00143 - Threading state (the event thread runs in the background)
00144 - Buffers for sending/receiving messages
00145 - Pending request tracking (for async request/response)
00146 - Statistics (for monitoring)
00147 - Authentication state (challenge/salt for password auth)
00148
00149 Why is it opaque (hidden in the .c file)?  So we can change the internal structure
00150 without breaking the API. Callers just use the pointer, they don't know what's inside.
00151
00152 Threading model:  Each connection has one background thread (event_thread) that
00153 processes WebSocket events, calls callbacks, etc.  The main application thread sends
00154 requests and gets responses.  This avoids the app freezing while waiting for responses.
00155
00156 Synchronization:  We use many mutexes because different parts of the connection
00157 are accessed from different threads:
00158 - state_mutex protects the connection state (so both threads see consistent state)
00159 - send_mutex protects sending (prevents two threads from sending at the same time)
00160 - requests_mutex protects the pending requests list
00161 - stats_mutex protects the statistics counters
00162 - scene_mutex protects the cached current scene name
00163
00164 The scene cache is an optimization - some operations need to know the current
00165 scene.  Instead of querying OBS every time, we cache it and update when we get
00166 SceneChanged events.
00167 */
```

```
00168
00169 struct obsws_connection {
00170     /* === Configuration and Setup === */
00171     obsws_config_t config;                  /* User-provided config (copied at construction) */
00172
00173     /* === Connection State === */
00174     obsws_state_t state;                    /* Current state (CONNECTED, CONNECTING, etc) */
00175     pthread_mutex_t state_mutex;            /* Protects state from concurrent access */
00176
00177     /* === WebSocket Layer === */
00178     struct lws_context *lws_context;        /* libwebsockets context (manages the WebSocket) */
00179     struct lws *wsi;                        /* WebSocket instance - the actual connection */
00180
00181     /* === Message Buffers ===
00182 We keep persistent buffers instead of allocating for every message because
00183 it's more efficient and avoids memory fragmentation.  */
00184     char *recv_buffer;                      /* Buffer for incoming messages from OBS */
00185     size_t recv_buffer_size;                /* Total capacity of receive buffer */
00186     size_t recv_buffer_used;                /* How many bytes are currently in the buffer */
00187
00188     char *send_buffer;                      /* Buffer for outgoing messages to OBS */
00189     size_t send_buffer_size;                /* Total capacity of send buffer */
00190
00191     /* === Background Thread ===
00192 The event thread continuously processes WebSocket events.  This allows the
00193 connection to receive messages and call callbacks without blocking the app.  */
00194     pthread_t event_thread;                 /* ID of the background thread */
00195     pthread_mutex_t send_mutex;             /* Prevents two threads from sending simultaneously */
00196     bool thread_running;                    /* Is the thread currently running?  */
00197     bool should_exit;                       /* Signal to thread:  time to stop */
00198
00199     /* === Async Request/Response Handling ===
00200 When you send a request, it returns immediately with a request ID. When the
00201 response comes back, we find the pending_request by ID and notify the waiter.  */
00202     pending_request_t *pending_requests;    /* Linked list of in-flight requests */
00203     pthread_mutex_t requests_mutex;         /* Protects the linked list */
00204
00205     /* === Performance Monitoring === */
00206     obsws_stats_t stats;                    /* Message counts, errors, latency, etc */
00207     pthread_mutex_t stats_mutex;            /* Protects stats from concurrent access */
00208
00209     /* === Keep-Alive / Health Monitoring ===
00210 We send periodic pings to detect when the connection dies.  If we don't get
00211 a pong back within the timeout, we know something is wrong.  */
00212     time_t last_ping_sent;                  /* When we last sent a ping */
00213     time_t last_pong_received;              /* When we last got a pong back */
00214
00215     /* === Reconnection ===
00216 If the connection drops and auto_reconnect is enabled, we try to reconnect.
00217 We use exponential backoff - each attempt waits longer, up to a maximum.  */
00218     uint32_t reconnect_attempts;            /* How many times have we tried reconnecting */
00219     uint32_t current_reconnect_delay;       /* How long we're waiting before next attempt */
00220
00221     /* === Authentication State ===
00222 OBS uses a challenge-response authentication scheme.  The server sends a
00223 challenge and salt, we compute a response using SHA256, and send it back.  */
00224     bool auth_required;                     /* Does OBS need authentication?  */
00225     char *challenge;                        /* Challenge string from OBS HELLO */
00226     char *salt;                             /* Salt string from OBS HELLO */
00227
00228     /* === Optimization Cache ===
00229 We cache the current scene to avoid querying OBS unnecessarily.  When we get
00230 a SceneChanged event, we update the cache.  */
00231     char *current_scene;                    /* Cached name of active scene */
00232     pthread_mutex_t scene_mutex;            /* Protects the cache */
00233 };
00234
00235 /* ============================================================================
00236 * Global State
00237 * ============================================================================ */
00238
00239 /* Global initialization flag - tracks whether obsws_init() has been called.
00240
00241 Why have global state at all?  Some underlying libraries (like libwebsockets
00242 and OpenSSL) need one-time initialization.  We do that in obsws_init() and
00243 make sure it only happens once, even if called multiple times.  This flag
00244 tracks whether we've done it.
00245
00246 We use a mutex to protect the flag because someone might call obsws_init()
00247 from multiple threads simultaneously.  The mutex ensures only one thread does
00248 the initialization.
00249 */
00250
00251 static bool g_library_initialized = false;      /* Have we called the init code yet?  */
00252 static obsws_log_level_t g_log_level = OBSWS_LOG_INFO;  /* Global filtering level */
00253 static obsws_debug_level_t g_debug_level = OBSWS_DEBUG_NONE;  /* Global debug verbosity */
00254 static pthread_mutex_t g_init_mutex = PTHREAD_MUTEX_INITIALIZER;  /* Thread-safe initialization */
```

```
00255
00256 /* ============================================================================
00257 * Logging
00258 * ============================================================================ */
00259
00260 /* Internal logging function - core logging infrastructure.
00261
00262 Design:  We filter by log level (higher level = more verbose).  If the message
00263 is below the current level, we don't even format it (saves CPU). If there's a
00264 user-provided callback, we use it; otherwise we print to stderr.
00265
00266 Why two parameters (conn and format)?  So we can log from both the main thread
00267 (with a connection object) and the global initialization code (without one).
00268 */
00269
00270 static void obsws_log(obsws_connection_t *conn, obsws_log_level_t level, const char *format, ...)  {
00271     /* Early exit if this message is too verbose */
00272     if (level > g_log_level) {
00273         return;
00274     }
00275
00276     /* Format the message using printf-style arguments */
00277     char message[1024];
00278     va_list args;
00279     va_start(args, format);
00280     vsnprintf(message, sizeof(message), format, args);
00281     va_end(args);
00282
00283     /* Route to user callback or stderr.  The callback lets the user handle logging
00284 however they want - write to file, send to logging service, etc.  */
00285     if (conn && conn->config.log_callback) {
00286         conn->config.log_callback(level, message, conn->config.user_data);
00287     } else {
00288         const char *level_str[] = {"NONE", "ERROR", "WARN", "INFO", "DEBUG"};
00289         fprintf(stderr, "[OBSWS-%s] %s\n", level_str[level], message);
00290     }
00291 }
00292
00293 /* Debug logging - finer control for protocol-level troubleshooting.
00294
00295 Separate from regular logging because debug messages are very verbose and
00296 developers typically only enable them when debugging specific issues.  The
00297 debug level goes 0-3, with higher levels including all output from lower levels.
00298
00299 We use a larger buffer (4KB) because debug messages can include JSON payloads.
00300 */
00301
00302 static void obsws_debug(obsws_connection_t *conn, obsws_debug_level_t min_level, const char *format,
    ...)  {
00303     /* Only output if global debug level is at or above the minimum for this message */
00304     if (g_debug_level < min_level) {
00305         return;
00306     }
00307
00308     /* Format with a larger buffer for JSON and other verbose output */
00309     char message[4096];
00310     va_list args;
00311     va_start(args, format);
00312     vsnprintf(message, sizeof(message), format, args);
00313     va_end(args);
00314
00315     /* Route through the callback as DEBUG-level logs */
00316     if (conn && conn->config.log_callback) {
00317         conn->config.log_callback(OBSWS_LOG_DEBUG, message, conn->config.user_data);
00318     } else {
00319         const char *debug_level_str[] = {"NONE", "LOW", "MED", "HIGH"};
00320         fprintf(stderr, "[DEBUG-%s] %s\n", debug_level_str[min_level], message);
00321     }
00322 }
00323
00324 /* ============================================================================
00325 * Utility Functions
00326 * ============================================================================ */
00327
00328 /* Generate a UUID v4 for request identification.
00329
00330 UUIDs uniquely identify each request, so when a response comes back, we can match
00331 it to the original request.  We use UUID v4 (random) because it's simple and the
00332 uniqueness probability is astronomically high.
00333
00334 Note:  This implementation uses rand() for simplicity.  A production system might
00335 use /dev/urandom for better randomness, but the current approach is fine for
00336 most use cases.  The protocol doesn't require cryptographically secure randomness.
00337
00338 Format:  8-4-4-4-12 hex digits with dashes, exactly 36 characters.
00339 Example:  550e8400-e29b-41d4-a716-446655440000
00340
```

```
00341 The version bits (0x4) and variant bits (0x8-b) mark this as a v4 UUID.
00342 */
00343
00344 static void generate_uuid(char *uuid_out) {
00345     unsigned int r1 = rand();
00346     unsigned int r2 = rand();
00347     unsigned int r3 = rand();
00348     unsigned int r4 = rand();
00349
00350     sprintf(uuid_out, "%08x-%04x-%04x-%04x-%04x%08x",
00351             r1,                          /* 8 hex digits */
00352             r2 & 0xFFFF,                 /* 4 hex digits */
00353             (r3 & 0x0FFF) | 0x4000,      /* 4 hex digits (set version 4) */
00354             (r4 & 0x3FFF) | 0x8000,      /* 4 hex digits (set variant bits) */
00355             rand() & 0xFFFF,             /* 4 hex digits */
00356             (unsigned int)rand());       /* 8 hex digits */
00357 }
00358
00359 /* Base64 encode binary data using OpenSSL.
00360
00361 Why base64 and not hex?  Hex would be twice as large.  Base64 is a standard
00362 encoding for binary data in text contexts (like WebSocket JSON messages).
00363
00364 We use OpenSSL's BIO (Basic I/O) interface for encoding because it's robust
00365 and well-tested.  The BIO_FLAGS_BASE64_NO_NL flag removes newlines that OpenSSL
00366 normally adds for readability - we don't want those in JSON.
00367 */
00368
00369 static char* base64_encode(const unsigned char *input, size_t length) {
00370     BIO *bio, *b64;
00371     BUF_MEM *buffer_ptr;
00372
00373     /* Set up OpenSSL base64 encoder:  b64 filter pushing to memory buffer */
00374     b64 = BIO_new(BIO_f_base64());
00375     bio = BIO_new(BIO_s_mem());
00376     bio = BIO_push(b64, bio);
00377
00378     /* Disable newlines in output (OpenSSL adds them by default for readability) */
00379     BIO_set_flags(bio, BIO_FLAGS_BASE64_NO_NL);
00380
00381     /* Encode the data */
00382     BIO_write(bio, input, length);
00383     BIO_flush(bio);
00384     BIO_get_mem_ptr(bio, &buffer_ptr);
00385
00386     /* Copy result to our own allocated buffer and null-terminate */
00387     char *result = malloc(buffer_ptr->length + 1);
00388     memcpy(result, buffer_ptr->data, buffer_ptr->length);
00389     result[buffer_ptr->length] = '\0';
00390
00391     BIO_free_all(bio);
00392     return result;
00393 }
00394
00395 /* Compute SHA256 hash of a null-terminated string.
00396
00397 SHA256 is a cryptographic hash function.  It's deterministic (same input always
00398 produces same output) and has an avalanche property (changing one bit in the
00399 input completely changes the output).  This makes it perfect for authentication
00400 protocols.
00401
00402 Why SHA256 instead of SHA1 or MD5?  SHA256 is current-best-practice.  SHA1 has
00403 known collisions, and MD5 is even more broken.  SHA256 is secure for the
00404 foreseeable future.
00405
00406 Why use EVP (Envelope) API instead of raw SHA256 functions?  EVP is higher-level
00407 and more flexible - if we ever need to support different hash algorithms, we
00408 just change one line.
00409 */
00410
00411 static void sha256_hash(const char *input, unsigned char *output) {
00412     EVP_MD_CTX *ctx = EVP_MD_CTX_new();
00413     EVP_DigestInit_ex(ctx, EVP_sha256(), NULL);
00414     EVP_DigestUpdate(ctx, input, strlen(input));
00415     EVP_DigestFinal_ex(ctx, output, NULL);
00416     EVP_MD_CTX_free(ctx);
00417 }
00418
00419 /* Generate OBS WebSocket v5 authentication response using challenge-response protocol.
00420
00421 OBS WebSocket v5 uses a two-step authentication protocol:
00422 1.  Server sends challenge + salt
00423 2.  Client computes:  secret = base64(sha256(password + salt))
00424 3.  Client computes:  response = base64(sha256(secret + challenge))
00425 4.  Client sends response
00426 5.  Server verifies by computing the same thing
00427
```

```
00428 Why this design?  The password never travels over the network.  Instead, a hash
00429 derived from the password (the secret) is combined with a fresh challenge each
00430 time, preventing replay attacks.  This is similar to HTTP Digest Authentication.
00431
00432 Why not use the password directly?  That would be incredibly insecure.  The
00433 two-step approach means an eavesdropper who sees the response can't use it
00434 again - the challenge was random and won't repeat.
00435 */
00436
00437 static char* generate_auth_response(const char *password, const char *salt, const char *challenge) {
00438     unsigned char secret_hash[SHA256_DIGEST_LENGTH];
00439     unsigned char auth_hash[SHA256_DIGEST_LENGTH];
00440
00441     /* Step 1:  Compute secret = base64(sha256(password + salt)) */
00442     char *password_salt = malloc(strlen(password) + strlen(salt) + 1);
00443     sprintf(password_salt, "%s%s", password, salt);
00444     sha256_hash(password_salt, secret_hash);
00445     free(password_salt);
00446
00447     char *secret = base64_encode(secret_hash, SHA256_DIGEST_LENGTH);
00448
00449     /* Step 2:  Compute auth response = base64(sha256(secret + challenge)) */
00450     char *secret_challenge = malloc(strlen(secret) + strlen(challenge) + 1);
00451     sprintf(secret_challenge, "%s%s", secret, challenge);
00452     sha256_hash(secret_challenge, auth_hash);
00453     free(secret_challenge);
00454     free(secret);
00455
00456     /* Return the final response, base64-encoded */
00457     char *auth_response = base64_encode(auth_hash, SHA256_DIGEST_LENGTH);
00458     return auth_response;
00459 }
00460
00461 /* ============================================================================
00462 * State Management
00463 * ============================================================================ */
00464
00465 /* Update connection state and notify callback if state changed.
00466
00467 This function is responsible for state transitions and notifying the user.
00468 We lock the mutex, make the change, unlock it, then call the callback without
00469 holding the lock.  Why release the lock before calling the callback?  Because
00470 the callback might take a long time, and we don't want to hold a lock during
00471 that time - it would prevent other threads from checking the state.
00472
00473 We only call the callback if the state actually changed.  This prevents spurious
00474 notifications if something tries to set the same state again.
00475 */
00476
00477 static void set_connection_state(obsws_connection_t *conn, obsws_state_t new_state) {
00478     /* Acquire lock, save old state, set new state, release lock */
00479     pthread_mutex_lock(&conn->state_mutex);
00480     obsws_state_t old_state = conn->state;
00481     conn->state = new_state;
00482     pthread_mutex_unlock(&conn->state_mutex);
00483
00484     /* Call callback only if state actually changed (not a duplicate) */
00485     if (old_state != new_state && conn->config.state_callback) {
00486         conn->config.state_callback(conn, old_state, new_state, conn->config.user_data);
00487     }
00488
00489     /* Log the transition for debugging/monitoring */
00490     obsws_log(conn, OBSWS_LOG_INFO, "State changed:  %s -> %s",
00491               obsws_state_string(old_state), obsws_state_string(new_state));
00492 }
00493
00494 /* ============================================================================
00495 * Request Management
00496 * ============================================================================ */
00497
00498 /* Create a new pending request and add it to the tracking list.
00499
00500 When we send a request to OBS, we need to track it so we can match the response
00501 when it arrives.  This function creates a pending_request_t struct and adds it
00502 to the linked list.  The request is initialized with the ID, a condition variable
00503 for waiting, and a current timestamp for timeout detection.
00504 */
00505
00506 static pending_request_t* create_pending_request(obsws_connection_t *conn, const char *request_id) {
00507     pending_request_t *req = calloc(1, sizeof(pending_request_t));
00508     if (!req) return NULL;
00509
00510     /* Copy request ID and ensure null termination */
00511     strncpy(req->request_id, request_id, OBSWS_UUID_LENGTH - 1);
00512     req->request_id[OBSWS_UUID_LENGTH - 1] = '\0';
00513
00514     /* Initialize request structure */
```

```
00515     req->response = calloc(1, sizeof(obsws_response_t));
00516     req->completed = false;
00517     req->timestamp = time(NULL);
00518     pthread_mutex_init(&req->mutex, NULL);
00519     pthread_cond_init(&req->cond, NULL);
00520
00521     /* Add to linked list of pending requests */
00522     pthread_mutex_lock(&conn->requests_mutex);
00523     req->next = conn->pending_requests;
00524     conn->pending_requests = req;
00525     pthread_mutex_unlock(&conn->requests_mutex);
00526
00527     return req;
00528 }
00529
00530 /* Find a pending request by its UUID */
00531 static pending_request_t* find_pending_request(obsws_connection_t *conn, const char *request_id) {
00532     pthread_mutex_lock(&conn->requests_mutex);
00533     pending_request_t *req = conn->pending_requests;
00534
00535     /* Search linked list for matching request ID */
00536     while (req) {
00537         if (strcmp(req->request_id, request_id) == 0) {
00538             pthread_mutex_unlock(&conn->requests_mutex);
00539             return req;
00540         }
00541         req = req->next;
00542     }
00543
00544     pthread_mutex_unlock(&conn->requests_mutex);
00545     return NULL;
00546 }
00547
00548 /* Remove a pending request from the tracking list and free it */
00549 static void remove_pending_request(obsws_connection_t *conn, pending_request_t *target) {
00550     pthread_mutex_lock(&conn->requests_mutex);
00551     pending_request_t **req = &conn->pending_requests;
00552
00553     /* Find and remove from linked list */
00554     while (*req) {
00555         if (*req == target) {
00556             *req = target->next;
00557             pthread_mutex_unlock(&conn->requests_mutex);
00558
00559             /* Clean up request resources */
00560             pthread_mutex_destroy(&target->mutex);
00561             pthread_cond_destroy(&target->cond);
00562             free(target);
00563             return;
00564         }
00565         req = &(*req)->next;
00566     }
00567
00568     pthread_mutex_unlock(&conn->requests_mutex);
00569 }
00570
00571 /* Clean up requests that have exceeded the timeout period */
00572 static void cleanup_old_requests(obsws_connection_t *conn) {
00573     time_t now = time(NULL);
00574     pthread_mutex_lock(&conn->requests_mutex);
00575
00576     pending_request_t **req = &conn->pending_requests;
00577     while (*req) {
00578         /* Check if request has timed out (30 seconds) */
00579         if (now - (*req)->timestamp > 30) {
00580             pending_request_t *old = *req;
00581             *req = old->next;
00582
00583             /* Mark as completed with timeout error */
00584             pthread_mutex_lock(&old->mutex);
00585             old->completed = true;
00586             old->response->success = false;
00587             old->response->error_message = strdup("Request timeout");
00588             pthread_cond_broadcast(&old->cond);  /* Wake waiting threads */
00589             pthread_mutex_unlock(&old->mutex);
00590         } else {
00591             req = &(*req)->next;
00592         }
00593     }
00594
00595     pthread_mutex_unlock(&conn->requests_mutex);
00596 }
00597
00598 /* ============================================================================
00599 * WebSocket Protocol Handling
00600 * ============================================================================ */
00601
```

```
00602 /**
00603 * @brief Handle the initial HELLO handshake message from OBS.
00604 *
00605 * When we first connect to OBS, the server sends a HELLO message containing
00606 * protocol version information and, if required, an authentication challenge
00607 * and salt.  This function extracts that information and immediately responds
00608 * with an IDENTIFY message.
00609 *
00610 * The authentication flow (if enabled) works as follows:
00611 * 1.  Server sends HELLO with a random challenge string and a salt
00612 * 2.  We compute:  secret = base64(SHA256(password + salt))
00613 * 3.  We compute:  response = base64(SHA256(secret + challenge))
00614 * 4.  We send this response in the IDENTIFY message
00615 * 5.  If it matches what the server computed, auth succeeds
00616 *
00617 * This challenge-response approach has several advantages over sending the
00618 * raw password:
00619 * - Password never travels across the network (only the computed response)
00620 * - If someone captures the network traffic, they can't replay the captured
00621 *   response because it's specific to this challenge (which was random)
00622 * - Similar to HTTP Digest Authentication (RFC 2617) but simpler
00623 *
00624 * The function transitions the connection state from CONNECTING to AUTHENTICATING,
00625 * sends the IDENTIFY message, and logs any authentication requirements.
00626 *
00627 * @param conn The connection structure containing buffers, config, and state
00628 * @param data The parsed JSON "d" field from the HELLO message, containing
00629 *             authentication info if auth is required
00630 * @return 0 on success, -1 on error (though errors are logged and connection
00631 *         continues - failure to authenticate will be detected by the server
00632 *         refusing to transition to IDENTIFIED state)
00633 *
00634 * @internal
00635 */
00636 static int handle_hello_message(obsws_connection_t *conn, cJSON *data) {
00637     /* DEBUG_LOW: Basic connection event */
00638     obsws_debug(conn, OBSWS_DEBUG_LOW, "Received Hello message from OBS");
00639
00640     cJSON *auth = cJSON_GetObjectItem(data, "authentication");
00641     if (auth) {
00642         conn->auth_required = true;
00643
00644         cJSON *challenge = cJSON_GetObjectItem(auth, "challenge");
00645         cJSON *salt = cJSON_GetObjectItem(auth, "salt");
00646
00647         if (challenge && salt) {
00648             conn->challenge = strdup(challenge->valuestring);
00649             conn->salt = strdup(salt->valuestring);
00650             /* DEBUG_MEDIUM: Show auth parameters */
00651             obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Authentication required - salt:  %s, challenge:
    %s",
00652                         conn->salt, conn->challenge);
00653         }
00654     } else {
00655         conn->auth_required = false;
00656         obsws_debug(conn, OBSWS_DEBUG_LOW, "No authentication required");
00657     }
00658
00659     /* Send Identify message */
00660     set_connection_state(conn, OBSWS_STATE_AUTHENTICATING);
00661
00662     cJSON *identify = cJSON_CreateObject();
00663     cJSON_AddNumberToObject(identify, "op", OBSWS_OPCODE_IDENTIFY);
00664
00665     cJSON *identify_data = cJSON_CreateObject();
00666     cJSON_AddNumberToObject(identify_data, "rpcVersion", OBSWS_PROTOCOL_VERSION);
00667     cJSON_AddNumberToObject(identify_data, "eventSubscriptions", OBSWS_EVENT_ALL);
00668
00669     if (conn->auth_required && conn->config.password) {
00670         /* DEBUG_HIGH: Show password being used */
00671         obsws_debug(conn, OBSWS_DEBUG_HIGH, "Generating auth response with password:  '%s'",
    conn->config.password);
00672         char *auth_response = generate_auth_response(conn->config.password, conn->salt,
    conn->challenge);
00673         /* DEBUG_MEDIUM: Show generated auth string */
00674         obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Generated auth response:  '%s'", auth_response);
00675         cJSON_AddStringToObject(identify_data, "authentication", auth_response);
00676         free(auth_response);
00677     } else {
00678         if (conn->auth_required) {
00679             obsws_log(conn, OBSWS_LOG_ERROR, "Authentication required but no password provided!");
00680         }
00681     }
00682
00683     cJSON_AddItemToObject(identify, "d", identify_data);
00684
00685     char *message = cJSON_PrintUnformatted(identify);
```

```
00686     cJSON_Delete(identify);
00687
00688     /* DEBUG_HIGH: Show full Identify message */
00689     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending Identify message:  %s", message);
00690
00691     pthread_mutex_lock(&conn->send_mutex);
00692     size_t len = strlen(message);
00693     if (len < conn->send_buffer_size - LWS_PRE) {
00694         memcpy(conn->send_buffer + LWS_PRE, message, len);
00695         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
      LWS_WRITE_TEXT);
00696         /* DEBUG_HIGH: Show bytes sent */
00697         obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sent %d bytes (requested %zu)", written, len);
00698     } else {
00699         obsws_log(conn, OBSWS_LOG_ERROR, "Message too large for send buffer:  %zu bytes", len);
00700     }
00701     pthread_mutex_unlock(&conn->send_mutex);
00702
00703     free(message);
00704     return 0;
00705 }
00706
00707 /**
00708 * @brief Handle the IDENTIFIED confirmation message from OBS.
00709 *
00710 * After we send an IDENTIFY message with authentication (or without if auth
00711 * isn't required), OBS responds with an IDENTIFIED message to confirm that
00712 * the connection is established and ready for commands.  This function marks
00713 * the connection as fully established, records connection statistics, and
00714 * resets the reconnection state.
00715 *
00716 * Receiving this message means:
00717 * - Authentication succeeded (if it was required)
00718 * - The server has accepted our protocol version
00719 * - We can now send REQUEST messages and receive EVENT messages
00720 * - The connection is in a healthy state
00721 *
00722 * We take this opportunity to:
00723 * 1.  Log successful authentication
00724 * 2.  Transition state to CONNECTED (the only valid way to enter this state)
00725 * 3.  Record the timestamp of successful connection (for statistics)
00726 * 4.  Reset the reconnection attempt counter and delay (we're connected!)
00727 *
00728 * @param conn The connection structure to mark as identified
00729 * @param data Unused (the IDENTIFIED message typically has no data payload)
00730 * @return Always 0 (this message type should never fail)
00731 *
00732 * @internal
00733 */
00734 static int handle_identified_message(obsws_connection_t *conn, cJSON *data) {
00735     (void)data;  /* Unused parameter */
00736     obsws_log(conn, OBSWS_LOG_INFO, "Successfully authenticated with OBS");
00737     /* DEBUG_LOW: Authentication success */
00738     obsws_debug(conn, OBSWS_DEBUG_LOW, "Identified message received - authentication successful");
00739     set_connection_state(conn, OBSWS_STATE_CONNECTED);
00740
00741     pthread_mutex_lock(&conn->stats_mutex);
00742     conn->stats.connected_since = time(NULL);
00743     conn->stats.reconnect_count = conn->reconnect_attempts;
00744     pthread_mutex_unlock(&conn->stats_mutex);
00745
00746     conn->reconnect_attempts = 0;
00747     conn->current_reconnect_delay = conn->config.reconnect_delay_ms;
00748
00749     return 0;
00750 }
00751
00752 /**
00753 * @brief Handle EVENT messages from OBS (real-time notifications).
00754 *
00755 * OBS continuously sends EVENT messages whenever something happens in the
00756 * application (scene changes, source muted/unmuted, recording started, etc.).
00757 * These events are only delivered if we subscribed to them in the IDENTIFY
00758 * message using the eventSubscriptions bitmask.
00759 *
00760 * This function:
00761 * 1.  Extracts the event type and event data from the JSON
00762 * 2.  Calls the user's event_callback if one was configured (async event loop)
00763 * 3.  Updates internal caches (e.g., current scene name on SceneChanged)
00764 *
00765 * Important threading note:  This is called from the background event_thread,
00766 * NOT from the main application thread.  Therefore:
00767 * - The event_callback is executed in the event thread context
00768 * - The callback should not block (keep processing fast)
00769 * - The callback should not make blocking library calls
00770 * - The event_data_str parameter is temporary and freed after callback returns
00771 * - If the callback needs to keep the data, it must copy it
```

```
00772 *
00773 * Scene caching optimization:
00774 * We maintain a cache of the currently active scene name in the connection
00775 * structure.  When we see a CurrentProgramSceneChanged event, we update this
00776 * cache immediately.  This avoids the need for the application to call
00777 * obsws_send_request(..., "GetCurrentProgramScene", ...)  repeatedly.
00778 * The cache is protected by scene_mutex for thread safety.
00779 *
00780 * @param conn The connection that received the event
00781 * @param data The parsed JSON "d" field containing eventType and eventData
00782 * @return 0 on success, -1 if the event data is malformed
00783 *
00784 * @internal
00785 */
00786 static int handle_event_message(obsws_connection_t *conn, cJSON *data) {
00787     cJSON *event_type = cJSON_GetObjectItem(data, "eventType");
00788     cJSON *event_data = cJSON_GetObjectItem(data, "eventData");
00789
00790     /* DEBUG_MEDIUM: Show event type */
00791     if (event_type) {
00792         obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Event received:  %s", event_type->valuestring);
00793     }
00794
00795     if (event_type && conn->config.event_callback) {
00796         char *event_data_str = event_data ?  cJSON_PrintUnformatted(event_data) :  NULL;
00797         /* DEBUG_HIGH: Show full event data */
00798         if (event_data_str) {
00799             obsws_debug(conn, OBSWS_DEBUG_HIGH, "Event data:  %s", event_data_str);
00800         }
00801         conn->config.event_callback(conn, event_type->valuestring, event_data_str,
      conn->config.user_data);
00802         if (event_data_str) free(event_data_str);
00803     }
00804
00805     /* Update current scene cache if scene changed */
00806     if (event_type && strcmp(event_type->valuestring, "CurrentProgramSceneChanged") == 0) {
00807         cJSON *scene_name = cJSON_GetObjectItem(event_data, "sceneName");
00808         if (scene_name) {
00809             pthread_mutex_lock(&conn->scene_mutex);
00810             free(conn->current_scene);
00811             conn->current_scene = strdup(scene_name->valuestring);
00812             pthread_mutex_unlock(&conn->scene_mutex);
00813             /* DEBUG_LOW: Scene changes are important */
00814             obsws_debug(conn, OBSWS_DEBUG_LOW, "Scene changed to:  %s", scene_name->valuestring);
00815         }
00816     }
00817
00818     return 0;
00819 }
00820
00821 /**
00822 * @brief Handle REQUEST_RESPONSE messages from OBS (responses to our commands).
00823 *
00824 * When we send a REQUEST message (via obsws_send_request), OBS processes it
00825 * and sends back a REQUEST_RESPONSE message with the same requestId that we
00826 * used.  This function matches the response to the pending request, populates
00827 * the response data, and wakes up the waiting thread.
00828 *
00829 * The async request/response pattern allows the application to send multiple
00830 * requests without waiting for each response.  The flow is:
00831 * 1.  Application calls obsws_send_request("GetScenes", ...)  -> returns immediately
00832 * 2.  The request is created with a unique UUID and added to pending_requests list
00833 * 3.  Background thread sends the request to OBS
00834 * 4.  Background thread waits for response (on condition variable, not busy-polling)
00835 * 5.  OBS responds with REQUEST_RESPONSE containing the requestId
00836 * 6.  This function matches it to the pending request
00837 * 7.  Function sets response->success, response->status_code, response->response_data
00838 * 8.  Function signals the condition variable to wake the waiting thread
00839 * 9.  Application thread wakes up with the response ready
00840 *
00841 * This is far more efficient than synchronous request/response because:
00842 * - No blocking wait in the background thread
00843 * - Multiple requests can be in-flight simultaneously
00844 * - Doesn't freeze the application while waiting for OBS
00845 * - Condition variables are more efficient than polling
00846 *
00847 * Response structure contains:
00848 * - success:  Did the operation succeed?  (not the HTTP status, but "was it valid?")
00849 * - status_code:  The OBS response code (0 = success, >0 = error)
00850 * - response_data:  JSON string with the actual result (e.g., scene list)
00851 * - error_message:  If something failed, what was the reason?
00852 *
00853 * @param conn The connection that received the response
00854 * @param data The parsed JSON "d" field containing requestId, requestStatus, etc.
00855 * @return 0 on success, -1 if the response is malformed (e.g., missing requestId)
00856 *
00857 * @internal
```

```
00858 */
00859 static int handle_request_response_message(obsws_connection_t *conn, cJSON *data) {
00860     cJSON *request_id = cJSON_GetObjectItem(data, "requestId");
00861     if (!request_id) return -1;
00862 
00863     /* DEBUG_MEDIUM: Show request ID being processed */
00864     obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Response received for request:  %s",
      request_id->valuestring);
00865 
00866     pending_request_t *req = find_pending_request(conn, request_id->valuestring);
00867     if (!req) {
00868         obsws_log(conn, OBSWS_LOG_WARNING, "Received response for unknown request:  %s",
      request_id->valuestring);
00869         return -1;
00870     }
00871 
00872     pthread_mutex_lock(&req->mutex);
00873 
00874     cJSON *request_status = cJSON_GetObjectItem(data, "requestStatus");
00875     if (request_status) {
00876         cJSON *result = cJSON_GetObjectItem(request_status, "result");
00877         cJSON *code = cJSON_GetObjectItem(request_status, "code");
00878         cJSON *comment = cJSON_GetObjectItem(request_status, "comment");
00879 
00880         req->response->success = result ?  result->valueint :  false;
00881         req->response->status_code = code ?  code->valueint :  -1;
00882 
00883         if (comment) {
00884             req->response->error_message = strdup(comment->valuestring);
00885         }
00886     }
00887 
00888     cJSON *response_data = cJSON_GetObjectItem(data, "responseData");
00889     if (response_data) {
00890         req->response->response_data = cJSON_PrintUnformatted(response_data);
00891     }
00892 
00893     req->completed = true;
00894     pthread_cond_broadcast(&req->cond);
00895     pthread_mutex_unlock(&req->mutex);
00896 
00897     return 0;
00898 }
00899 
00900 /**
00901 * @brief Route incoming WebSocket messages to appropriate handlers based on opcode.
00902 *
00903 * Every message from OBS contains an "op" field (opcode) that identifies the
00904 * message type.  This function:
00905 * 1.  Parses the JSON to extract the opcode and data
00906 * 2.  Routes to the appropriate handler function based on the opcode
00907 * 3.  Updates statistics (messages_received, bytes_received)
00908 *
00909 * The OBS WebSocket protocol uses these opcodes:
00910 * - HELLO (0):  Server greeting with auth info - handled by handle_hello_message
00911 * - IDENTIFY (1):  Client auth - we send this, don't receive it
00912 * - IDENTIFIED (2):  Auth success - handled by handle_identified_message
00913 * - EVENT (5):  Real-time notifications - handled by handle_event_message
00914 * - REQUEST_RESPONSE (7):  Command responses - handled by handle_request_response_message
00915 * - Other opcodes like REIDENTIFY, batch operations:  not currently handled
00916 *
00917 * This is one of the most critical functions in the library because it's in
00918 * the hot path of message processing.  Performance matters here.  We keep it
00919 * lightweight and defer heavy processing to the specific handlers.
00920 *
00921 * Error handling is conservative:  malformed JSON or missing opcode doesn't
00922 * crash the connection, it just logs and continues.  This allows us to be
00923 * resilient to protocol variations or corruption.
00924 *
00925 * @param conn The connection that received the message
00926 * @param message Pointer to the raw message data (not null-terminated)
00927 * @param len Number of bytes in the message
00928 * @return 0 on success, -1 on parse error (does not disconnect)
00929 *
00930 * @internal
00931 */
00932 static int handle_websocket_message(obsws_connection_t *conn, const char *message, size_t len) {
00933     /* DEBUG_HIGH: Show full message content */
00934     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Received message (%zu bytes):  %.*s", len, (int)len,
      message);
00935 
00936     cJSON *json = cJSON_ParseWithLength(message, len);
00937     if (!json) {
00938         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to parse JSON message");
00939         return -1;
00940     }
00941 
```

```
00942     cJSON *op = cJSON_GetObjectItem(json, "op");
00943     cJSON *data = cJSON_GetObjectItem(json, "d");
00944
00945     if (!op) {
00946         obsws_log(conn, OBSWS_LOG_ERROR, "Message missing 'op' field");
00947         cJSON_Delete(json);
00948         return -1;
00949     }
00950
00951     /* DEBUG_MEDIUM: Show opcode being processed */
00952     obsws_debug(conn, OBSWS_DEBUG_MEDIUM, "Processing opcode:  %d", op->valueint);
00953
00954     int result = 0;
00955     switch (op->valueint) {
00956         case OBSWS_OPCODE_HELLO:
00957             result = handle_hello_message(conn, data);
00958             break;
00959         case OBSWS_OPCODE_IDENTIFIED:
00960             result = handle_identified_message(conn, data);
00961             break;
00962         case OBSWS_OPCODE_EVENT:
00963             result = handle_event_message(conn, data);
00964             break;
00965         case OBSWS_OPCODE_REQUEST_RESPONSE:
00966             result = handle_request_response_message(conn, data);
00967             break;
00968         default:
00969             obsws_log(conn, OBSWS_LOG_DEBUG, "Unhandled opcode:  %d", op->valueint);
00970             break;
00971     }
00972
00973     cJSON_Delete(json);
00974
00975     pthread_mutex_lock(&conn->stats_mutex);
00976     conn->stats.messages_received++;
00977     conn->stats.bytes_received += len;
00978     pthread_mutex_unlock(&conn->stats_mutex);
00979
00980     return result;
00981 }
00982
00983 /* ============================================================================
00984 * libwebsockets Callbacks
00985 * ============================================================================ */
00986
00987 /**
00988 * @brief libwebsockets callback - routes WebSocket events to our handlers.
00989 *
00990 * libwebsockets is an event-driven WebSocket library.  It calls this callback
00991 * function whenever something happens (connection established, data received,
00992 * connection closed, etc.).  The "reason" parameter identifies what happened.
00993 *
00994 * We handle these key reasons:
00995 * - LWS_CALLBACK_CLIENT_ESTABLISHED: TCP/WebSocket handshake complete, ready for messages
00996 * - LWS_CALLBACK_CLIENT_RECEIVE: Data arrived from OBS
00997 * - LWS_CALLBACK_CLIENT_WRITEABLE: Socket is writable (less common with our design)
00998 * - LWS_CALLBACK_CLIENT_CONNECTION_ERROR: Connection failed (network error, bad host, etc.)
00999 * - LWS_CALLBACK_CLIENT_CLOSED: Connection closed normally
01000 * - LWS_CALLBACK_WSI_DESTROY: Cleanup callback
01001 *
01002 * Important:  This callback is called from the background event_thread, not
01003 * the main application thread.  So it must be thread-safe and not block.
01004 *
01005 * Message assembly:  OBS WebSocket messages might arrive fragmented (multiple
01006 * packets).  We accumulate them in recv_buffer and check lws_is_final_fragment()
01007 * to know when a complete message has arrived.  Only then do we parse it.
01008 *
01009 * Error handling:  Connection errors and receive buffer overflows are logged
01010 * but don't crash.  We just transition to ERROR state and let the connection
01011 * cleanup/reconnection logic handle recovery.
01012 *
01013 * @param wsi The WebSocket instance (provided by libwebsockets)
01014 * @param reason The callback reason (LWS_CALLBACK_*)
01015 * @param user Our connection pointer (registered at context creation)
01016 * @param in Incoming data (for LWS_CALLBACK_CLIENT_RECEIVE)
01017 * @param len Size of incoming data
01018 * @return 0 for success, -1 for error (affects connection handling)
01019 *
01020 * @internal
01021 */
01022 static int lws_callback(struct lws *wsi, enum lws_callback_reasons reason,
01023                         void *user, void *in, size_t len) {
01024     obsws_connection_t *conn = (obsws_connection_t *)user;
01025
01026     switch (reason) {
01027         case LWS_CALLBACK_CLIENT_ESTABLISHED:
01028             obsws_log(conn, OBSWS_LOG_INFO, "WebSocket connection established");
```

```
01029             set_connection_state(conn, OBSWS_STATE_CONNECTING);
01030             break;
01031
01032         case LWS_CALLBACK_CLIENT_RECEIVE:
01033             if (conn->recv_buffer_used + len < conn->recv_buffer_size) {
01034                 memcpy(conn->recv_buffer + conn->recv_buffer_used, in, len);
01035                 conn->recv_buffer_used += len;
01036
01037                 if (lws_is_final_fragment(wsi)) {
01038                     handle_websocket_message(conn, conn->recv_buffer, conn->recv_buffer_used);
01039                     conn->recv_buffer_used = 0;
01040                 }
01041             } else {
01042                 obsws_log(conn, OBSWS_LOG_ERROR, "Receive buffer overflow");
01043                 conn->recv_buffer_used = 0;
01044             }
01045             break;
01046
01047         case LWS_CALLBACK_CLIENT_WRITEABLE:
01048             /* Handle queued sends if needed */
01049             break;
01050
01051         case LWS_CALLBACK_CLIENT_CONNECTION_ERROR:
01052             obsws_log(conn, OBSWS_LOG_ERROR, "Connection error:  %s", in ?  (char *)in :  "unknown");
01053             set_connection_state(conn, OBSWS_STATE_ERROR);
01054             break;
01055
01056         case LWS_CALLBACK_CLIENT_CLOSED:
01057             obsws_log(conn, OBSWS_LOG_INFO, "WebSocket connection closed (reason in 'in' param)");
01058             if (in && len > 0) {
01059                 obsws_log(conn, OBSWS_LOG_INFO, "Close reason:  %.*s", (int)len, (char*)in);
01060             }
01061             set_connection_state(conn, OBSWS_STATE_DISCONNECTED);
01062             break;
01063
01064         case LWS_CALLBACK_WSI_DESTROY:
01065             conn->wsi = NULL;
01066             break;
01067
01068         default:
01069             break;
01070     }
01071
01072     return 0;
01073 }
01074
01075 static const struct lws_protocols protocols[] = {
01076     {
01077         "obs-websocket",
01078         lws_callback,
01079         0,
01080         OBSWS_DEFAULT_BUFFER_SIZE,
01081         0, /* id */
01082         NULL, /* user */
01083         0 /* tx_packet_size */
01084     },
01085     { NULL, NULL, 0, 0, 0, NULL, 0 }
01086 };
01087
01088 /* ========================================================================
01089 * Event Thread
01090 * ======================================================================== */
01091
01092 /**
01093 * @brief Background thread function that continuously processes WebSocket events.
01094 *
01095 * Each connection has one background thread dedicated to processing WebSocket
01096 * messages and timers.  The main application thread remains free to make requests
01097 * and do application work without blocking.
01098 *
01099 * This thread:
01100 * 1.  Calls lws_service() to pump the libwebsockets event loop (typically blocks
01101 *    for 50ms waiting for events, then processes them and returns)
01102 * 2.  Periodically cleans up old/timed-out requests
01103 * 3.  Sends keep-alive pings if configured (to detect dead connections)
01104 * 4.  Exits gracefully when should_exit flag is set
01105 *
01106 * Lifetime:  This thread is created in obsws_connect() and destroyed in
01107 * obsws_disconnect() using pthread_join().  The should_exit flag is used to
01108 * signal the thread to stop, which it checks at the start of each loop.
01109 *
01110 * The lws_service() call is the core of this loop.  It:
01111 * - Waits up to 50ms for data from the network using select/poll
01112 * - If data arrives, invokes lws_callback to notify us
01113 * - Returns after ~50ms even if no data (so we stay responsive to should_exit)
01114 *
01115 * This asynchronous design has several advantages:
```

```
01116 * - App thread isn't blocked waiting for responses
01117 * - Multiple requests can be in-flight simultaneously
01118 * - Events can be delivered to callbacks instantly (no polling delay)
01119 * - Automatic keep-alive pings keep the connection alive through firewalls
01120 *
01121 * Memory note:  Callbacks invoked from this thread have access to the same
01122 * connection object as the main thread, hence all the mutexes protecting
01123 * critical sections.  The pending_request_t condition variables synchronize
01124 * request responses between this thread and application threads.
01125 *
01126 * @param arg The obsws_connection_t* that this thread services
01127 * @return Always NULL (threads don't return values)
01128 *
01129 * @internal
01130 */
01131 static void* event_thread_func(void *arg) {
01132     obsws_connection_t *conn = (obsws_connection_t *)arg;
01133
01134     bool should_continue = true;
01135     while (should_continue) {
01136         /* Check exit flag with mutex protection */
01137         pthread_mutex_lock(&conn->state_mutex);
01138         should_continue = !conn->should_exit;
01139         pthread_mutex_unlock(&conn->state_mutex);
01140
01141         if (!should_continue) break;
01142
01143         if (conn->lws_context) {
01144             lws_service(conn->lws_context, 50);
01145
01146             /* Cleanup old requests periodically */
01147             cleanup_old_requests(conn);
01148
01149             /* Handle keep-alive pings */
01150             if (conn->config.ping_interval_ms > 0 && conn->state == OBSWS_STATE_CONNECTED) {
01151                 time_t now = time(NULL);
01152                 if (now - conn->last_ping_sent >= conn->config.ping_interval_ms / 1000) {
01153                     if (conn->wsi) {
01154                         lws_callback_on_writable(conn->wsi);
01155                     }
01156                     conn->last_ping_sent = now;
01157                 }
01158             }
01159         } else {
01160             struct timespec ts = {0, 50000000};  /* 50ms = 50,000,000 nanoseconds */
01161             nanosleep(&ts, NULL);
01162         }
01163     }
01164
01165     return NULL;
01166 }
01167
01168 /* ============================================================================
01169 * Public API Implementation
01170 * ============================================================================ */
01171
01172 /**
01173 * @brief Initialize the libwsv5 library.
01174 *
01175 * This function must be called before creating any connections.  It:
01176 * 1.  Initializes OpenSSL (EVP library for hashing)
01177 * 2.  Seeds the random number generator for UUID generation
01178 * 3.  Sets the global g_library_initialized flag
01179 *
01180 * Thread safety:  This function is thread-safe.  Multiple threads can call it
01181 * simultaneously, and only one will actually do the initialization (protected
01182 * by g_init_mutex).  Subsequent calls are no-ops.
01183 *
01184 * Note:  obsws_connect() will call this automatically if you forget, so you
01185 * don't *have* to call it explicitly.  But doing so allows you to initialize
01186 * in a controlled way, separate from connection creation.
01187 *
01188 * Cleanup:  When you're done with the library, call obsws_cleanup() to
01189 * deallocate resources.  This is technically optional on program exit (the OS
01190 * cleans up anyway), but good practice for testing and library shutdown.
01191 *
01192 * @return OBSWS_OK always (initialization cannot fail in the current design)
01193 *
01194 * @see obsws_cleanup, obsws_connect
01195 */
01196 obsws_error_t obsws_init(void) {
01197     pthread_mutex_lock(&g_init_mutex);
01198
01199     if (g_library_initialized) {
01200         pthread_mutex_unlock(&g_init_mutex);
01201         return OBSWS_OK;
01202     }
```

```
01203
01204     /* Initialize OpenSSL */
01205     OpenSSL_add_all_algorithms();
01206
01207     /* Seed random number generator */
01208     srand(time(NULL));
01209
01210     g_library_initialized = true;
01211     pthread_mutex_unlock(&g_init_mutex);
01212
01213     return OBSWS_OK;
01214 }
01215
01216 /**
01217 * @brief Clean up library resources.
01218 *
01219 * Call this when you're done with the library to deallocate OpenSSL resources.
01220 * This is a counterpart to obsws_init().
01221 *
01222 * Important:  Make sure all obsws_connection_t objects have been disconnected
01223 * and freed via obsws_disconnect() before calling this.  If not, you might have
01224 * dangling references and resource leaks.
01225 *
01226 * Thread safety:  This function is thread-safe and idempotent (safe to call
01227 * multiple times).  It checks g_library_initialized before doing anything.
01228 *
01229 * Note:  This is optional on program exit because the OS will clean up memory
01230 * anyway.  But it's good practice for:
01231 * - Library consumers that need clean shutdown
01232 * - Memory leak detectors / Valgrind tests
01233 * - Programs that unload the library
01234 *
01235 * @see obsws_init
01236 */
01237 void obsws_cleanup(void) {
01238     pthread_mutex_lock(&g_init_mutex);
01239
01240     if (!g_library_initialized) {
01241         pthread_mutex_unlock(&g_init_mutex);
01242         return;
01243     }
01244
01245     EVP_cleanup();
01246     g_library_initialized = false;
01247
01248     pthread_mutex_unlock(&g_init_mutex);
01249 }
01250
01251 /**
01252 * @brief Get the library version string.
01253 *
01254 * Returns a semantic version string like "1.0.0" that identifies which
01255 * version of libwsv5 is being used.  Useful for debugging and logging.
01256 *
01257 * @return Pointer to static version string (don't free)
01258 */
01259 const char* obsws_version(void) {
01260     return OBSWS_VERSION;
01261 }
01262
01263 /**
01264 * @brief Set the global log level threshold.
01265 *
01266 * The library logs various messages during operation.  This function sets which
01267 * messages are displayed.  All messages at the specified level and higher
01268 * severity are shown; lower severity messages are hidden.
01269 *
01270 * Levels in increasing severity:
01271 * - OBSWS_LOG_DEBUG: Low-level diagnostic info (too verbose for production)
01272 * - OBSWS_LOG_INFO: General informational messages (usual choice)
01273 * - OBSWS_LOG_WARNING: Potentially problematic situations (degraded but working)
01274 * - OBSWS_LOG_ERROR: Error conditions that need attention
01275 *
01276 * Example:  If you call obsws_set_log_level(OBSWS_LOG_WARNING), you'll see
01277 * only WARNING and ERROR messages, but not INFO or DEBUG messages.
01278 *
01279 * Thread safety:  This modifies a global variable without locking, so if you
01280 * might call this from multiple threads, use synchronization externally.
01281 *
01282 * @param level The minimum severity level to display
01283 * @see obsws_set_debug_level
01284 */
01285 void obsws_set_log_level(obsws_log_level_t level) {
01286     g_log_level = level;
01287 }
01288
01289 /**
```

```
01290 * @brief Set the global debug level.
01291 *
01292 * Debug logging is separate from regular logging.  It provides extremely
01293 * detailed trace information about the WebSocket protocol, message parsing,
01294 * authentication, etc.  This is useful during development and troubleshooting.
01295 *
01296 * Debug levels:
01297 * - OBSWS_DEBUG_NONE: No debug output (fastest)
01298 * - OBSWS_DEBUG_LOW: Major state transitions and connection events
01299 * - OBSWS_DEBUG_MEDIUM: Message types and handlers invoked
01300 * - OBSWS_DEBUG_HIGH: Full message content and every operation
01301 *
01302 * Debug logging is independent of log level.  You can have OBSWS_LOG_ERROR
01303 * set (hide non-error logs) but still see OBSWS_DEBUG_HIGH output.
01304 *
01305 * Performance warning:  OBSWS_DEBUG_HIGH produces enormous output and will
01306 * slow down the library significantly.  Only use during debugging!
01307 *
01308 * Thread safety:  Same as obsws_set_log_level (modifies global without locking).
01309 *
01310 * @param level The debug verbosity level
01311 * @see obsws_set_log_level, obsws_get_debug_level
01312 */
01313 void obsws_set_debug_level(obsws_debug_level_t level) {
01314     g_debug_level = level;
01315 }
01316
01317 /**
01318 * @brief Get the current debug level.
01319 *
01320 * This is a read-only query - it doesn't change anything, just returns the
01321 * current global debug level that was set by obsws_set_debug_level().
01322 *
01323 * Useful for conditional logging in your application, e.g.:
01324 * ```
01325 * if (obsws_get_debug_level() >= OBSWS_DEBUG_MEDIUM) {
01326 *     // do expensive trace operation
01327 * }
01328 * ```
01329 *
01330 * @return The currently active debug level
01331 * @see obsws_set_debug_level
01332 */
01333 obsws_debug_level_t obsws_get_debug_level(void) {
01334     return g_debug_level;
01335 }
01336
01337 /**
01338 * @brief Initialize a connection configuration structure with safe defaults.
01339 *
01340 * Before calling obsws_connect(), you create an obsws_config_t structure
01341 * with the connection parameters.  This function initializes that structure
01342 * with sensible defaults so you only need to change what's different for
01343 * your use case.
01344 *
01345 * Default values set:
01346 * - port:  4455 (OBS WebSocket v5 default port)
01347 * - use_ssl:  false (OBS uses ws://, not wss://)
01348 * - connect_timeout_ms:  5000 (5 seconds to connect)
01349 * - recv_timeout_ms:  5000 (5 seconds to receive each message)
01350 * - send_timeout_ms:  5000 (5 seconds to send each message)
01351 * - ping_interval_ms:  10000 (send ping every 10 seconds)
01352 * - ping_timeout_ms:  5000 (expect pong within 5 seconds)
01353 * - auto_reconnect:  true (reconnect automatically if connection drops)
01354 * - reconnect_delay_ms:  1000 (start with 1 second delay)
01355 * - max_reconnect_delay_ms:  30000 (max wait is 30 seconds)
01356 * - max_reconnect_attempts:  0 (infinite attempts)
01357 *
01358 * After calling this, you typically set:
01359 * - config.host = "localhost" (where OBS is running)
01360 * - config.password = "your_password" (if OBS has auth enabled)
01361 * - config.event_callback = your_callback_func (to receive events)
01362 *
01363 * @param config Pointer to structure to initialize (must not be NULL)
01364 *
01365 * @see obsws_connect
01366 */
01367 void obsws_config_init(obsws_config_t *config) {
01368     memset(config, 0, sizeof(obsws_config_t));
01369
01370     config->port = 4455;
01371     config->use_ssl = false;
01372     config->connect_timeout_ms = 5000;
01373     config->recv_timeout_ms = 5000;
01374     config->send_timeout_ms = 5000;
01375     config->ping_interval_ms = 10000;
01376     config->ping_timeout_ms = 5000;
```

```
01377     config->auto_reconnect = true;
01378     config->reconnect_delay_ms = 1000;
01379     config->max_reconnect_delay_ms = 30000;
01380     config->max_reconnect_attempts = 0; /* Infinite */
01381 }
01382
01383 /**
01384 * @brief Establish a connection to OBS.
01385 *
01386 * This is the main entry point for using the library.  You provide a configuration
01387 * structure (initialized with obsws_config_init and then customized), and this
01388 * function connects to OBS, authenticates if needed, and spawns a background
01389 * thread to handle incoming messages and events.
01390 *
01391 * The function returns immediately - it doesn't wait for the connection to
01392 * complete.  Instead, it:
01393 * 1.  Creates a connection structure with the provided config
01394 * 2.  Allocates buffers for sending and receiving messages
01395 * 3.  Creates a libwebsockets context and connects to OBS
01396 * 4.  Spawns a background event_thread to process WebSocket messages
01397 * 5.  Returns the connection handle
01398 *
01399 * Connection states:  The connection progresses through states:
01400 * - DISCONNECTED -> CONNECTING (TCP handshake, WebSocket upgrade)
01401 * - CONNECTING -> AUTHENTICATING (receive HELLO, send IDENTIFY)
01402 * - AUTHENTICATING -> CONNECTED (receive IDENTIFIED)
01403 *
01404 * You don't have to wait for CONNECTED state before calling obsws_send_request,
01405 * but requests sent while not connected will return OBSWS_ERROR_NOT_CONNECTED.
01406 *
01407 * Memory ownership:  The connection structure is allocated and owned by this
01408 * function.  You must free it by calling obsws_disconnect().  Don't free it directly
01409 * with free() - that will cause memory leaks (threads won't be cleaned up properly).
01410 *
01411 * Error cases:
01412 * - NULL config or config->host:  Returns NULL
01413 * - libwebsockets context creation fails:  Returns NULL and logs error
01414 * - Network connection fails:  Returns valid pointer but connection stays in ERROR state
01415 * - Bad password:  Returns valid pointer but stays in AUTHENTICATING (never reaches CONNECTED)
01416 *
01417 * Note:  This function calls obsws_init() automatically if the library isn't
01418 * already initialized.
01419 *
01420 * @param config Pointer to initialized obsws_config_t with connection parameters
01421 * @return Pointer to new connection handle, or NULL if creation failed
01422 *
01423 * @see obsws_disconnect, obsws_get_state, obsws_send_request
01424 */
01425 obsws_connection_t* obsws_connect(const obsws_config_t *config) {
01426     if (!g_library_initialized) {
01427         obsws_init();
01428     }
01429
01430     if (!config || !config->host) {
01431         return NULL;
01432     }
01433
01434     obsws_connection_t *conn = calloc(1, sizeof(obsws_connection_t));
01435     if (!conn) return NULL;
01436
01437     /* Copy configuration */
01438     memcpy(&conn->config, config, sizeof(obsws_config_t));
01439     if (config->host) conn->config.host = strdup(config->host);
01440     if (config->password) conn->config.password = strdup(config->password);
01441
01442     /* Initialize mutexes */
01443     pthread_mutex_init(&conn->state_mutex, NULL);
01444     pthread_mutex_init(&conn->send_mutex, NULL);
01445     pthread_mutex_init(&conn->requests_mutex, NULL);
01446     pthread_mutex_init(&conn->stats_mutex, NULL);
01447     pthread_mutex_init(&conn->scene_mutex, NULL);
01448
01449     /* Allocate buffers */
01450     conn->recv_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01451     conn->recv_buffer = malloc(conn->recv_buffer_size);
01452     conn->send_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01453     conn->send_buffer = malloc(conn->send_buffer_size);
01454
01455     conn->state = OBSWS_STATE_DISCONNECTED;
01456     conn->current_reconnect_delay = config->reconnect_delay_ms;
01457
01458     /* Create libwebsockets context */
01459     struct lws_context_creation_info info;
01460     memset(&info, 0, sizeof(info));
01461
01462     info.port = CONTEXT_PORT_NO_LISTEN;
01463     info.protocols = protocols;
```

```
01464     info.gid = -1;
01465     info.uid = -1;
01466     info.options = LWS_SERVER_OPTION_DO_SSL_GLOBAL_INIT;
01467 
01468     conn->lws_context = lws_create_context(&info);
01469     if (!conn->lws_context) {
01470         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to create libwebsockets context");
01471         free(conn->recv_buffer);
01472         free(conn->send_buffer);
01473         free(conn);
01474         return NULL;
01475     }
01476 
01477     /* Connect to OBS */
01478     struct lws_client_connect_info ccinfo;
01479     memset(&ccinfo, 0, sizeof(ccinfo));
01480 
01481     ccinfo.context = conn->lws_context;
01482     ccinfo.address = conn->config.host;
01483     ccinfo.port = conn->config.port;
01484     ccinfo.path = "/";
01485     ccinfo.host = ccinfo.address;
01486     ccinfo.origin = ccinfo.address;
01487     ccinfo.protocol = protocols[0].name;
01488     ccinfo.userdata = conn;
01489 
01490     if (config->use_ssl) {
01491         ccinfo.ssl_connection = LCCSCF_USE_SSL;
01492     }
01493 
01494     conn->wsi = lws_client_connect_via_info(&ccinfo);
01495     if (!conn->wsi) {
01496         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to initiate connection");
01497         lws_context_destroy(conn->lws_context);
01498         free(conn->recv_buffer);
01499         free(conn->send_buffer);
01500         free(conn);
01501         return NULL;
01502     }
01503 
01504     /* Start event thread - protect flags with mutex */
01505     pthread_mutex_lock(&conn->state_mutex);
01506     conn->thread_running = true;
01507     conn->should_exit = false;
01508     pthread_mutex_unlock(&conn->state_mutex);
01509 
01510     pthread_create(&conn->event_thread, NULL, event_thread_func, conn);
01511 
01512     obsws_log(conn, OBSWS_LOG_INFO, "Connecting to OBS at %s:%d", config->host, config->port);
01513 
01514     return conn;
01515 }
01516 
01517 /**
01518  * @brief Disconnect from OBS and clean up connection resources.
01519  *
01520  * This is the counterpart to obsws_connect().  It cleanly shuts down the connection,
01521  * stops the background event thread, and frees all allocated resources.
01522  *
01523  * The function performs these steps:
01524  * 1.  Signal the event_thread to stop by setting should_exit flag
01525  * 2.  Wait for the event_thread to actually exit using pthread_join()
01526  * 3.  Send a normal WebSocket close frame to OBS (if connected)
01527  * 4.  Destroy the libwebsockets context
01528  * 5.  Free all pending requests (they won't get responses now, but don't leak memory)
01529  * 6.  Free buffers, config, authentication data
01530  * 7.  Destroy all mutexes and condition variables
01531  * 8.  Free the connection structure
01532  *
01533  * After calling this, the connection pointer is invalid.  Don't use it again.
01534  *
01535  * Safe to call multiple times:  If you call it twice, the second call will be
01536  * a no-op (because conn will be NULL).
01537  *
01538  * Safe to call even if connection never fully established:  If you disconnect
01539  * while in CONNECTING or AUTHENTICATING state, everything is still cleaned up.
01540  *
01541  * Important:  This function blocks until the event_thread exits.  If you have
01542  * a callback that's blocked, this will deadlock.  Make sure your callbacks
01543  * don't block!
01544  *
01545  * @param conn The connection to close (can be NULL - safe to call)
01546  *
01547  * @see obsws_connect
01548  */
01549 void obsws_disconnect(obsws_connection_t *conn) {
01550     if (!conn) return;
```

```
01551
01552     obsws_log(conn, OBSWS_LOG_INFO, "Disconnecting from OBS");
01553
01554     /* Stop event thread - protect flag with mutex */
01555     pthread_mutex_lock(&conn->state_mutex);
01556     conn->should_exit = true;
01557     bool thread_was_running = conn->thread_running;
01558     pthread_mutex_unlock(&conn->state_mutex);
01559
01560     if (thread_was_running) {
01561         pthread_join(conn->event_thread, NULL);
01562     }
01563
01564     /* Close WebSocket - only if connected */
01565     if (conn->wsi && conn->state == OBSWS_STATE_CONNECTED) {
01566         lws_close_reason(conn->wsi, LWS_CLOSE_STATUS_NORMAL, NULL, 0);
01567     }
01568
01569     /* Cleanup libwebsockets */
01570     if (conn->lws_context) {
01571         lws_context_destroy(conn->lws_context);
01572     }
01573
01574     /* Free pending requests */
01575     pthread_mutex_lock(&conn->requests_mutex);
01576     pending_request_t *req = conn->pending_requests;
01577     while (req) {
01578         pending_request_t *next = req->next;
01579         if (req->response) {
01580             obsws_response_free(req->response);
01581         }
01582         pthread_mutex_destroy(&req->mutex);
01583         pthread_cond_destroy(&req->cond);
01584         free(req);
01585         req = next;
01586     }
01587     pthread_mutex_unlock(&conn->requests_mutex);
01588
01589     /* Free resources */
01590     free(conn->recv_buffer);
01591     free(conn->send_buffer);
01592     free((char *)conn->config.host);
01593     free((char *)conn->config.password);
01594     free(conn->challenge);
01595     free(conn->salt);
01596     free(conn->current_scene);
01597
01598     /* Destroy mutexes */
01599     pthread_mutex_destroy(&conn->state_mutex);
01600     pthread_mutex_destroy(&conn->send_mutex);
01601     pthread_mutex_destroy(&conn->requests_mutex);
01602     pthread_mutex_destroy(&conn->stats_mutex);
01603     pthread_mutex_destroy(&conn->scene_mutex);
01604
01605     free(conn);
01606 }
01607
01608 /**
01609 * @brief Check if a connection is actively connected to OBS.
01610 *
01611 * Convenience function that returns true if the connection is in CONNECTED state
01612 * and false otherwise.  Useful for checking before sending requests.
01613 *
01614 * Thread-safe:  This function locks the state_mutex before checking, so it's
01615 * safe to call from any thread.
01616 *
01617 * Return value:  The connection must be in OBSWS_STATE_CONNECTED to return true.
01618 * If it's CONNECTING, AUTHENTICATING, ERROR, or DISCONNECTED, this returns false.
01619 *
01620 * @param conn The connection to check (NULL is safe - returns false)
01621 * @return true if connected, false otherwise
01622 *
01623 * @see obsws_get_state
01624 */
01625 bool obsws_is_connected(const obsws_connection_t *conn) {
01626     if (!conn) return false;
01627
01628     /* Thread-safe state check */
01629     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
01630     bool connected = (conn->state == OBSWS_STATE_CONNECTED);
01631     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
01632
01633     return connected;
01634 }
01635
01636 /**
01637 * @brief Get the current connection state.
```

```
01638 *
01639 * Returns one of the connection states:
01640 * - OBSWS_STATE_DISCONNECTED: Not connected, idle
01641 * - OBSWS_STATE_CONNECTING: TCP connection established, waiting for WebSocket handshake
01642 * - OBSWS_STATE_AUTHENTICATING: WebSocket established, waiting for auth response
01643 * - OBSWS_STATE_CONNECTED: Connected and ready for requests
01644 * - OBSWS_STATE_ERROR: Connection encountered an error
01645 *
01646 * State transitions normally follow this flow:
01647 * DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED
01648 *
01649 * But with errors, you can also have:
01650 * (any state) -> ERROR
01651 *
01652 * You don't usually need to call this - just try to send requests and check
01653 * the error code.  But it's useful for monitoring and debugging.
01654 *
01655 * Thread-safe:  This function locks the state_mutex, so it's safe to call
01656 * from any thread.
01657 *
01658 * @param conn The connection to check (NULL is safe - returns DISCONNECTED)
01659 * @return The current connection state
01660 *
01661 * @see obsws_is_connected
01662 */
01663 obsws_state_t obsws_get_state(const obsws_connection_t *conn) {
01664     if (!conn) return OBSWS_STATE_DISCONNECTED;
01665
01666     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
01667     obsws_state_t state = conn->state;
01668     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
01669
01670     return state;
01671 }
01672
01673 /**
01674 * @brief Retrieve performance and connectivity statistics.
01675 *
01676 * The library maintains statistics about the connection:
01677 * - messages_sent / messages_received:  Count of WebSocket messages
01678 * - bytes_sent / bytes_received:  Total bytes transmitted/received
01679 * - connected_since:  Timestamp of when we reached CONNECTED state
01680 * - reconnect_count:  How many times we've reconnected (0 if never disconnected)
01681 *
01682 * These can be useful for:
01683 * - Monitoring connection health
01684 * - Detecting stalled connections (if bytes_received stops increasing)
01685 * - Debugging and performance profiling
01686 * - Health dashboards or logging
01687 *
01688 * Thread-safe:  This function acquires stats_mutex and copies the entire
01689 * stats structure, so it's safe to call from any thread.  The copy operation
01690 * is atomic from the caller's perspective.
01691 *
01692 * Example usage:
01693 * "'
01694 * obsws_stats_t stats;
01695 * obsws_get_stats(conn, &stats);
01696 * printf("Received %zu messages, %zu bytes\\n",
01697 *        stats.messages_received, stats.bytes_received);
01698 * "'
01699 *
01700 * @param conn The connection to query (NULL returns error)
01701 * @param stats Pointer to stats structure to fill (must not be NULL)
01702 * @return OBSWS_OK on success, OBSWS_ERROR_INVALID_PARAM if conn or stats is NULL
01703 *
01704 * @see obsws_stats_t
01705 */
01706 obsws_error_t obsws_get_stats(const obsws_connection_t *conn, obsws_stats_t *stats) {
01707     if (!conn || !stats) return OBSWS_ERROR_INVALID_PARAM;
01708
01709     pthread_mutex_lock((pthread_mutex_t *)&conn->stats_mutex);
01710     memcpy(stats, &conn->stats, sizeof(obsws_stats_t));
01711     pthread_mutex_unlock((pthread_mutex_t *)&conn->stats_mutex);
01712
01713     return OBSWS_OK;
01714 }
01715
01716 /**
01717 * @brief Send a synchronous request to OBS and wait for the response.
01718 *
01719 * This is the core function for all OBS operations.  It implements the asynchronous
01720 * request-response pattern of the OBS WebSocket v5 protocol:
01721 *
01722 * **Protocol Flow:**
01723 * 1.  Generate a unique UUID for this request (used to match responses)
01724 * 2.  Create a pending_request_t to track the in-flight operation
```

```
01725 * 3.  Build the request JSON with opcode 6 (REQUEST)
01726 * 4.  Send the message via lws_write()
01727 * 5.  Block the caller with pthread_cond_timedwait() until response arrives
01728 * 6.  Return the response to caller (who owns it and must free with obsws_response_free)
01729 *
01730 * **Why synchronous from caller's perspective?**
01731 * Although WebSocket messages are async at the protocol level, we provide a
01732 * synchronous API - the caller sends a request and blocks until the response
01733 * arrives.  This is simpler for application code than callback-based async APIs.
01734 *
01735 * Behind the scenes, the background event_thread continuously processes WebSocket
01736 * messages.  When a REQUEST_RESPONSE (opcode 7) arrives matching a pending request
01737 * ID, it signals the waiting condition variable, waking up the blocked caller.
01738 *
01739 * **Performance implications:**
01740 * - Thread-safe:  The main app thread can be blocked in obsws_send_request() while
01741 *   the background event_thread processes other messages
01742 * - No polling:  Uses condition variables, not CPU-wasting polling loops
01743 * - Can make multiple simultaneous requests from different threads (up to
01744 *   OBSWS_MAX_PENDING_REQUESTS = 256)
01745 *
01746 * **Example usage:**
01747 * ```
01748 * obsws_response_t *response = NULL;
01749 * obsws_error_t err = obsws_send_request(conn, "SetCurrentProgramScene",
01750 *                                        "{\"sceneName\":  \"Scene1\"}",
01751 *                                        &response, 0);
01752 * if (err == OBSWS_OK && response && response->success) {
01753 *     printf("Scene switched successfully\\n");
01754 * }
01755 * obsws_response_free(response);
01756 * ```
01757 *
01758 * @param conn Connection object (must be in CONNECTED state)
01759 * @param request_type OBS request type like "GetCurrentProgramScene", "SetCurrentProgramScene", etc.
01760 * @param request_data Optional JSON string with request parameters.  NULL for no parameters.
01761 *                     Example:  "{\"sceneName\":  \"Scene1\"}"
01762 * @param response Output pointer for the response.  Will be allocated by this function.
01763 *                 Caller must free with obsws_response_free().  Can be NULL if caller doesn't
01764 *                 need the response (but response is still consumed from server).
01765 * @param timeout_ms Timeout in milliseconds (0 = use config->recv_timeout_ms, typically 30000ms)
01766 *
01767 * @return OBSWS_OK if response received (check response->success for operation success)
01768 * @return OBSWS_ERROR_INVALID_PARAM if conn, request_type, or response pointer is NULL
01769 * @return OBSWS_ERROR_NOT_CONNECTED if connection is not in CONNECTED state
01770 * @return OBSWS_ERROR_OUT_OF_MEMORY if pending request allocation fails
01771 * @return OBSWS_ERROR_SEND_FAILED if message send fails (buffer too small, invalid wsi, etc)
01772 * @return OBSWS_ERROR_TIMEOUT if no response received within timeout_ms
01773 *
01774 * @see obsws_response_t, obsws_response_free, obsws_error_string
01775 */
01776 obsws_error_t obsws_send_request(obsws_connection_t *conn, const char *request_type,
01777                                  const char *request_data, obsws_response_t **response, uint32_t
      timeout_ms) {
01778     if (!conn || !request_type || !response) {
01779         return OBSWS_ERROR_INVALID_PARAM;
01780     }
01781
01782     if (conn->state != OBSWS_STATE_CONNECTED) {
01783         return OBSWS_ERROR_NOT_CONNECTED;
01784     }
01785
01786     /* Generate request ID */
01787     char request_id[OBSWS_UUID_LENGTH];
01788     generate_uuid(request_id);
01789
01790     /* Create pending request */
01791     pending_request_t *req = create_pending_request(conn, request_id);
01792     if (!req) {
01793         return OBSWS_ERROR_OUT_OF_MEMORY;
01794     }
01795
01796     /* Build request JSON */
01797     cJSON *request = cJSON_CreateObject();
01798     cJSON_AddNumberToObject(request, "op", OBSWS_OPCODE_REQUEST);
01799
01800     cJSON *d = cJSON_CreateObject();
01801     cJSON_AddStringToObject(d, "requestType", request_type);
01802     cJSON_AddStringToObject(d, "requestId", request_id);
01803
01804     if (request_data) {
01805         cJSON *data = cJSON_Parse(request_data);
01806         if (data) {
01807             cJSON_AddItemToObject(d, "requestData", data);
01808         }
01809     }
01810
```

```
01811     cJSON_AddItemToObject(request, "d", d);
01812
01813     char *message = cJSON_PrintUnformatted(request);
01814     cJSON_Delete(request);
01815
01816     /* DEBUG_HIGH: Show request being sent */
01817     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending request (ID: %s):  %s", request_id, message);
01818
01819     /* Send request */
01820     pthread_mutex_lock(&conn->send_mutex);
01821     size_t len = strlen(message);
01822     obsws_error_t result = OBSWS_OK;
01823
01824     if (len < conn->send_buffer_size - LWS_PRE && conn->wsi) {
01825         memcpy(conn->send_buffer + LWS_PRE, message, len);
01826         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
      LWS_WRITE_TEXT);
01827
01828         if (written < 0) {
01829             result = OBSWS_ERROR_SEND_FAILED;
01830         } else {
01831             pthread_mutex_lock(&conn->stats_mutex);
01832             conn->stats.messages_sent++;
01833             conn->stats.bytes_sent += len;
01834             pthread_mutex_unlock(&conn->stats_mutex);
01835         }
01836     } else {
01837         result = OBSWS_ERROR_SEND_FAILED;
01838     }
01839     pthread_mutex_unlock(&conn->send_mutex);
01840
01841     free(message);
01842
01843     if (result != OBSWS_OK) {
01844         remove_pending_request(conn, req);
01845         return result;
01846     }
01847
01848     /* Wait for response */
01849     if (timeout_ms == 0) {
01850         timeout_ms = conn->config.recv_timeout_ms;
01851     }
01852
01853     struct timespec ts;
01854     clock_gettime(CLOCK_REALTIME, &ts);
01855     ts.tv_sec += timeout_ms / 1000;
01856     ts.tv_nsec += (timeout_ms % 1000) * 1000000;
01857     if (ts.tv_nsec >= 1000000000) {
01858         ts.tv_sec++;
01859         ts.tv_nsec -= 1000000000;
01860     }
01861
01862     pthread_mutex_lock(&req->mutex);
01863     while (!req->completed) {
01864         int wait_result = pthread_cond_timedwait(&req->cond, &req->mutex, &ts);
01865         if (wait_result == ETIMEDOUT) {
01866             pthread_mutex_unlock(&req->mutex);
01867             remove_pending_request(conn, req);
01868             return OBSWS_ERROR_TIMEOUT;
01869         }
01870     }
01871
01872     *response = req->response;
01873     req->response = NULL; /* Transfer ownership */
01874     pthread_mutex_unlock(&req->mutex);
01875
01876     remove_pending_request(conn, req);
01877
01878     return OBSWS_OK;
01879 }
01880
01881 /**
01882  * @brief Switch OBS to a specific scene.
01883  *
01884  * This is a high-level convenience function for scene switching.  It demonstrates several
01885  * important design patterns in the library:
01886  *
01887  * **Optimization:  Scene Cache**
01888  * Before sending a request to OBS, this function checks the cached current scene.  If the
01889  * requested scene is already active, it returns immediately without network overhead.
01890  * The cache is maintained by the event thread when SceneChanged events arrive.
01891  *
01892  * **Memory Ownership**
01893  * If the caller provides response pointer, they receive ownership and must call
01894  * obsws_response_free().  If response is NULL, the function frees the response internally.
01895  * This flexibility allows three usage patterns:
01896  * 1.  Check response:  `obsws_set_current_scene(conn, name, &resp); if (resp->success) ...`
```

```
01897 * 2.  Ignore response:  `obsws_set_current_scene(conn, name, NULL);` (response is freed internally)
01898 * 3.  Just check error:  `if (obsws_send_request(...)  != OBSWS_OK) ...`
01899 *
01900 * **Example usage:**
01901 * ```
01902 * // Pattern 1:  Check response details
01903 * obsws_response_t *response = NULL;
01904 * if (obsws_set_current_scene(conn, "Scene1", &response) == OBSWS_OK &&
01905 *     response && response->success) {
01906 *     printf("Switched successfully\\n");
01907 * } else {
01908 *     printf("Switch failed:  %s\\n", response ?  response->error_message :  "unknown");
01909 * }
01910 * obsws_response_free(response);
01911 *
01912 * // Pattern 2:  Ignore response (simpler)
01913 * obsws_set_current_scene(conn, "Scene1", NULL);
01914 * ```
01915 *
01916 * **Thread-safety:**
01917 * - Scene cache is protected by scene_mutex
01918 * - Safe to call from any thread
01919 * - Multiple calls can happen simultaneously (each uses send_mutex)
01920 *
01921 * @param conn Connection object (must be in CONNECTED state)
01922 * @param scene_name Name of the scene to switch to.  Must not be NULL.
01923 * @param response Optional output for response details.  If provided, caller owns it
01924 *                 and must free with obsws_response_free().  If NULL, response is
01925 *                 freed internally.
01926 *
01927 * @return OBSWS_OK if request sent and response received (check response->success
01928 *         for whether the scene switch actually succeeded in OBS)
01929 * @return OBSWS_ERROR_INVALID_PARAM if conn or scene_name is NULL
01930 * @return OBSWS_ERROR_NOT_CONNECTED if connection not ready
01931 * @return OBSWS_ERROR_TIMEOUT if no response from OBS
01932 *
01933 * @see obsws_get_current_scene, obsws_response_t, obsws_response_free
01934 */
01935 obsws_error_t obsws_set_current_scene(obsws_connection_t *conn, const char *scene_name,
      obsws_response_t **response) {
01936     if (!conn || !scene_name) {
01937         return OBSWS_ERROR_INVALID_PARAM;
01938     }
01939
01940     /* Check cache to avoid redundant switches */
01941     pthread_mutex_lock(&conn->scene_mutex);
01942     bool already_current = (conn->current_scene && strcmp(conn->current_scene, scene_name) == 0);
01943     pthread_mutex_unlock(&conn->scene_mutex);
01944
01945     if (already_current) {
01946         obsws_log(conn, OBSWS_LOG_DEBUG, "Already on scene:  %s", scene_name);
01947         if (response) {
01948             *response = calloc(1, sizeof(obsws_response_t));
01949             (*response)->success = true;
01950         }
01951         return OBSWS_OK;
01952     }
01953
01954     cJSON *request_data = cJSON_CreateObject();
01955     cJSON_AddStringToObject(request_data, "sceneName", scene_name);
01956
01957     char *data_str = cJSON_PrintUnformatted(request_data);
01958     cJSON_Delete(request_data);
01959
01960     obsws_response_t *resp = NULL;
01961     obsws_error_t result = obsws_send_request(conn, "SetCurrentProgramScene", data_str, &resp, 0);
01962     free(data_str);
01963
01964     if (result == OBSWS_OK && resp && resp->success) {
01965         pthread_mutex_lock(&conn->scene_mutex);
01966         free(conn->current_scene);
01967         conn->current_scene = strdup(scene_name);
01968         pthread_mutex_unlock(&conn->scene_mutex);
01969
01970         obsws_log(conn, OBSWS_LOG_INFO, "Switched to scene:  %s", scene_name);
01971     }
01972
01973     if (response) {
01974         *response = resp;
01975     } else if (resp) {
01976         obsws_response_free(resp);
01977     }
01978
01979     return result;
01980 }
01981
01982 /**
```

```
01983 * @brief Query the currently active scene in OBS.
01984 *
01985 * This function queries OBS for the active scene name and returns it in the provided
01986 * buffer.  It also updates the local scene cache to keep it synchronized.
01987 *
01988 * **Cache Synchronization**
01989 * This function always queries the OBS server (doesn't use cached value).  When the
01990 * response arrives, it updates the cache.  This ensures the library's cached scene
01991 * name stays in sync with the actual OBS state.
01992 *
01993 * **Buffer Management**
01994 * The caller provides a buffer.  If the scene name is longer than buffer_size-1,
01995 * it will be truncated and null-terminated.  Always check the returned buffer length
01996 * if you need the full name.
01997 *
01998 * **Thread-safety**
01999 * The scene_mutex protects the cache update, so this is safe to call from any thread.
02000 * Multiple concurrent calls are safe but will all query OBS (no deduplication).
02001 *
02002 * **Example usage:**
02003 * ```
02004 * char scene[256];
02005 * if (obsws_get_current_scene(conn, scene, sizeof(scene)) == OBSWS_OK) {
02006 *     printf("Current scene:  %s\\n", scene);
02007 * }
02008 * ```
02009 *
02010 * @param conn Connection object (must be in CONNECTED state)
02011 * @param scene_name Buffer to receive the scene name (must not be NULL)
02012 * @param buffer_size Size of scene_name buffer (must be > 0)
02013 *
02014 * @return OBSWS_OK if scene name retrieved successfully
02015 * @return OBSWS_ERROR_INVALID_PARAM if conn, scene_name is NULL or buffer_size is 0
02016 * @return OBSWS_ERROR_NOT_CONNECTED if connection not ready
02017 * @return OBSWS_ERROR_TIMEOUT if OBS doesn't respond
02018 * @return OBSWS_ERROR_PARSE_FAILED if response can't be parsed
02019 *
02020 * @see obsws_set_current_scene
02021 */
02022 obsws_error_t obsws_get_current_scene(obsws_connection_t *conn, char *scene_name, size_t buffer_size)
      {
02023     if (!conn || !scene_name || buffer_size == 0) {
02024         return OBSWS_ERROR_INVALID_PARAM;
02025     }
02026
02027     obsws_response_t *response = NULL;
02028     obsws_error_t result = obsws_send_request(conn, "GetCurrentProgramScene", NULL, &response, 0);
02029
02030     if (result == OBSWS_OK && response && response->success && response->response_data) {
02031         cJSON *data = cJSON_Parse(response->response_data);
02032         if (data) {
02033             cJSON *name = cJSON_GetObjectItem(data, "currentProgramSceneName");
02034             if (name && name->valuestring) {
02035                 strncpy(scene_name, name->valuestring, buffer_size - 1);
02036                 scene_name[buffer_size - 1] = '\0';
02037
02038                 /* Update cache */
02039                 pthread_mutex_lock(&conn->scene_mutex);
02040                 free(conn->current_scene);
02041                 conn->current_scene = strdup(name->valuestring);
02042                 pthread_mutex_unlock(&conn->scene_mutex);
02043             }
02044             cJSON_Delete(data);
02045         }
02046     }
02047
02048     if (response) {
02049         obsws_response_free(response);
02050     }
02051
02052     return result;
02053 }
02054
02055 /**
02056 * @brief Start recording in OBS.
02057 *
02058 * Tells OBS to begin recording the current scene composition to disk.  The recording
02059 * path and format are determined by OBS settings, not by this library.
02060 *
02061 * This is a convenience wrapper around obsws_send_request() using the OBS
02062 * "StartRecord" request type.
02063 *
02064 * **Return value interpretation:**
02065 * - OBSWS_OK: Request was sent and OBS responded (check response->success)
02066 * - Other errors:  Network/connection problem
02067 *
02068 * **Example usage:**
```

```
02069 * ```
02070 * obsws_response_t *resp = NULL;
02071 * if (obsws_start_recording(conn, &resp) == OBSWS_OK && resp && resp->success) {
02072 *     printf("Recording started\\n");
02073 * }
02074 * obsws_response_free(resp);
02075 * ```
02076 *
02077 * @param conn Connection object (must be in CONNECTED state)
02078 * @param response Optional output for response.  Caller owns if provided, must free.
02079 *
02080 * @return OBSWS_OK if response received (check response->success for success)
02081 * @return OBSWS_ERROR_NOT_CONNECTED if connection not ready
02082 * @return OBSWS_ERROR_TIMEOUT if OBS doesn't respond
02083 *
02084 * @see obsws_stop_recording, obsws_send_request, obsws_response_free
02085 */
02086 obsws_error_t obsws_start_recording(obsws_connection_t *conn, obsws_response_t **response) {
02087     return obsws_send_request(conn, "StartRecord", NULL, response, 0);
02088 }
02089
02090 /**
02091 * @brief Stop recording in OBS.
02092 *
02093 * Tells OBS to stop the currently active recording.  If no recording is in progress,
02094 * OBS returns success anyway (idempotent operation).
02095 *
02096 * This is a convenience wrapper around obsws_send_request() using the OBS
02097 * "StopRecord" request type.
02098 *
02099 * **Example usage:**
02100 * ```
02101 * if (obsws_stop_recording(conn, NULL) != OBSWS_OK) {
02102 *     fprintf(stderr, "Failed to stop recording\\n");
02103 * }
02104 * ```
02105 *
02106 * @param conn Connection object (must be in CONNECTED state)
02107 * @param response Optional output for response.  Caller owns if provided, must free.
02108 *
02109 * @return OBSWS_OK if response received (check response->success for success)
02110 * @return OBSWS_ERROR_NOT_CONNECTED if connection not ready
02111 * @return OBSWS_ERROR_TIMEOUT if OBS doesn't respond
02112 *
02113 * @see obsws_start_recording, obsws_send_request, obsws_response_free
02114 */
02115 obsws_error_t obsws_stop_recording(obsws_connection_t *conn, obsws_response_t **response) {
02116     return obsws_send_request(conn, "StopRecord", NULL, response, 0);
02117 }
02118
02119 /**
02120 * @brief Start streaming in OBS.
02121 *
02122 * Tells OBS to begin streaming to the configured destination (Twitch, YouTube, etc).
02123 * The stream settings (URL, key, bitrate, etc) are determined by OBS settings.
02124 *
02125 * This is a convenience wrapper around obsws_send_request() using the OBS
02126 * "StartStream" request type.
02127 *
02128 * **Thread-safe:** Safe to call from any thread while connected.
02129 *
02130 * @param conn Connection object (must be in CONNECTED state)
02131 * @param response Optional output for response.  Caller owns if provided, must free.
02132 *
02133 * @return OBSWS_OK if response received (check response->success for success)
02134 * @return OBSWS_ERROR_NOT_CONNECTED if connection not ready
02135 * @return OBSWS_ERROR_TIMEOUT if OBS doesn't respond
02136 *
02137 * @see obsws_stop_streaming, obsws_send_request, obsws_response_free
02138 */
02139 obsws_error_t obsws_start_streaming(obsws_connection_t *conn, obsws_response_t **response) {
02140     return obsws_send_request(conn, "StartStream", NULL, response, 0);
02141 }
02142
02143 /**
02144 * @brief Stop streaming in OBS.
02145 *
02146 * Tells OBS to stop the active stream.  If not currently streaming, OBS returns
02147 * success anyway (idempotent operation).
02148 *
02149 * This is a convenience wrapper around obsws_send_request() using the OBS
02150 * "StopStream" request type.
02151 *
02152 * @param conn Connection object (must be in CONNECTED state)
02153 * @param response Optional output for response.  Caller owns if provided, must free.
02154 *
02155 * @return OBSWS_OK if response received (check response->success for success)
```

```
02156 * @return OBSWS_ERROR_NOT_CONNECTED if connection not ready
02157 * @return OBSWS_ERROR_TIMEOUT if OBS doesn't respond
02158 *
02159 * @see obsws_start_streaming, obsws_send_request, obsws_response_free
02160 */
02161 obsws_error_t obsws_stop_streaming(obsws_connection_t *conn, obsws_response_t **response) {
02162     return obsws_send_request(conn, "StopStream", NULL, response, 0);
02163 }
02164
02165 /**
02166 * @brief Free a response object previously allocated by obsws_send_request().
02167 *
02168 * This function safely deallocates all memory associated with a response:
02169 * - The error_message string (if present)
02170 * - The response_data JSON string (if present)
02171 * - The response structure itself
02172 *
02173 * **Safe to call with NULL**
02174 * Calling with NULL is safe and does nothing - no crash or error.
02175 * This allows for simpler cleanup:
02176 * "'
02177 * obsws_response_t *resp = NULL;
02178 * obsws_send_request(..., &resp, ...);
02179 * obsws_response_free(resp);  // Safe even if send_request failed
02180 * "'
02181 *
02182 * **Memory ownership**
02183 * - obsws_send_request() allocates the response - you must free it
02184 * - High-level functions like obsws_set_current_scene() can optionally take
02185 *   response ownership or free internally based on parameters
02186 *
02187 * **NOT thread-safe**
02188 * Each response should only be accessed/freed by one thread.  If multiple threads
02189 * need the response, use higher-level synchronization.
02190 *
02191 * @param response Response to free.  Can be NULL (does nothing if so).
02192 *
02193 * @see obsws_send_request, obsws_response_t
02194 */
02195 void obsws_response_free(obsws_response_t *response) {
02196     if (!response) return;
02197
02198     free(response->error_message);
02199     free(response->response_data);
02200     free(response);
02201 }
02202
02203 /**
02204 * @brief Convert an error code to a human-readable string.
02205 *
02206 * Utility function for error reporting and logging.  Returns a brief English
02207 * description of each error code.
02208 *
02209 * **Never returns NULL**
02210 * Unknown error codes return "Unknown error", so it's always safe to use
02211 * the returned pointer without NULL checks.
02212 *
02213 * **Example usage:**
02214 * "'
02215 * obsws_error_t err = obsws_send_request(...);
02216 * if (err != OBSWS_OK) {
02217 *     fprintf(stderr, "Error:  %s\\n", obsws_error_string(err));
02218 * }
02219 * "'
02220 *
02221 * **Strings are constants**
02222 * Returned strings are statically allocated - do not modify or free them.
02223 *
02224 * @param error The error code to describe
02225 * @return Pointer to a static string describing the error
02226 *
02227 * @see obsws_error_t
02228 */
02229 const char* obsws_error_string(obsws_error_t error) {
02230     switch (error) {
02231         case OBSWS_OK:  return "Success";
02232         case OBSWS_ERROR_INVALID_PARAM:  return "Invalid parameter";
02233         case OBSWS_ERROR_CONNECTION_FAILED:  return "Connection failed";
02234         case OBSWS_ERROR_AUTH_FAILED:  return "Authentication failed";
02235         case OBSWS_ERROR_TIMEOUT:  return "Timeout";
02236         case OBSWS_ERROR_SEND_FAILED:  return "Send failed";
02237         case OBSWS_ERROR_RECV_FAILED:  return "Receive failed";
02238         case OBSWS_ERROR_PARSE_FAILED:  return "Parse failed";
02239         case OBSWS_ERROR_NOT_CONNECTED:  return "Not connected";
02240         case OBSWS_ERROR_ALREADY_CONNECTED:  return "Already connected";
02241         case OBSWS_ERROR_OUT_OF_MEMORY:  return "Out of memory";
02242         case OBSWS_ERROR_SSL_FAILED:  return "SSL failed";
```

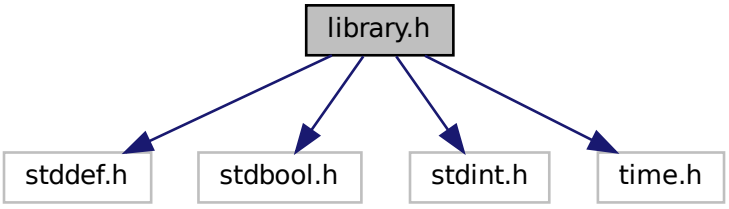
```
02243         default:  return "Unknown error";
02244     }
02245 }
02246
02247 /**
02248 * @brief Convert a connection state to a human-readable string.
02249 *
02250 * Utility function for logging and debugging.  Returns a brief English description
02251 * of each connection state.
02252 *
02253 * **State Transitions:**
02254 * - DISCONNECTED: Initial state or after disconnect()
02255 * - CONNECTING: connect() called, establishing TCP connection
02256 * - AUTHENTICATING: TCP connected, performing challenge-response auth
02257 * - CONNECTED: Auth succeeded, ready for requests
02258 * - ERROR: Network error or protocol failure, should reconnect
02259 *
02260 * **Valid transitions:**
02261 * ```
02262 * DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED
02263 * CONNECTED -> DISCONNECTED (on explicit disconnect)
02264 * CONNECTED -> ERROR (on network failure)
02265 * ERROR -> CONNECTING (on reconnect attempt)
02266 * ```
02267 *
02268 * **Example usage:**
02269 * ```
02270 * obsws_state_t state = obsws_get_state(conn);
02271 * printf("Connection state:  %s\\n", obsws_state_string(state));
02272 * ```
02273 *
02274 * Never returns NULL - unknown states return "Unknown".
02275 * Returned strings are static - do not modify or free.
02276 *
02277 * @param state The connection state to describe
02278 * @return Pointer to a static string describing the state
02279 *
02280 * @see obsws_get_state, obsws_state_t, obsws_is_connected
02281 */
02282 const char* obsws_state_string(obsws_state_t state) {
02283     switch (state) {
02284         case OBSWS_STATE_DISCONNECTED:  return "Disconnected";
02285         case OBSWS_STATE_CONNECTING:  return "Connecting";
02286         case OBSWS_STATE_AUTHENTICATING:  return "Authenticating";
02287         case OBSWS_STATE_CONNECTED:  return "Connected";
02288         case OBSWS_STATE_ERROR:  return "Error";
02289         default:  return "Unknown";
02290     }
02291 }
02292
02293 /**
02294 * @brief Process pending WebSocket events (compatibility function).
02295 *
02296 * This function is provided for API compatibility with single-threaded applications.
02297 * However, the libwsv5 library uses a background event_thread by default, so this
02298 * function is usually not needed - events are processed automatically in the background.
02299 *
02300 * **Background Event Processing:**
02301 * By design, all WebSocket messages (events, responses, etc.)  are processed by the
02302 * background event_thread.  This thread:
02303 * - Continuously calls lws_service() to pump WebSocket events
02304 * - Receives incoming messages from OBS
02305 * - Routes responses to waiting callers via condition variables
02306 * - Calls event callbacks for real-time events
02307 * - Maintains keep-alive pings
02308 *
02309 * Applications don't need to call this function - the thread handles everything.
02310 *
02311 * **What this function does:**
02312 * Currently, this is mainly for API compatibility.  It:
02313 * - Validates the connection object
02314 * - If timeout_ms > 0, sleeps for that duration
02315 * - Returns 0 (success)
02316 *
02317 * **When to use:**
02318 * - Most applications:  Don't call this - use background thread
02319 * - If you disable background thread:  Call this in your main loop
02320 *
02321 * **Example (not typical - background thread is recommended):**
02322 * ```
02323 * // Not recommended - background thread is better
02324 * while (app_running) {
02325 *     obsws_process_events(conn, 100);  // Check every 100ms
02326 * }
02327 * ```
02328 *
02329 * Better approach - let background thread handle it:
```

```
02330 * ```
02331 * obsws_connect(conn, "localhost", 4455, "password");
02332 * // Background thread processes events automatically
02333 * while (app_running) {
02334 *     // Your application code - no need to call process_events
02335 * }
02336 * obsws_disconnect(conn);
02337 * ```
02338 *
02339 * @param conn Connection object (can be NULL - returns error)
02340 * @param timeout_ms Sleep duration in milliseconds (0 = don't sleep)
02341 *
02342 * @return 0 on success
02343 * @return OBSWS_ERROR_INVALID_PARAM if conn is NULL
02344 *
02345 * @see obsws_connect, obsws_disconnect, event_thread_func
02346 */
02347 int obsws_process_events(obsws_connection_t *conn, uint32_t timeout_ms) {
02348     if (!conn) return OBSWS_ERROR_INVALID_PARAM;
02349
02350     /* Events are processed in the background thread */
02351     /* This function is provided for API compatibility */
02352     if (timeout_ms > 0) {
02353         struct timespec ts;
02354         ts.tv_sec = timeout_ms / 1000;
02355         ts.tv_nsec = (timeout_ms % 1000) * 1000000;
02356         nanosleep(&ts, NULL);
02357     }
02358
02359     return 0;
02360 }
```

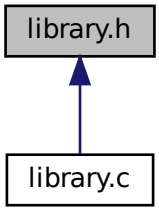
4.3 library.h File Reference

```
#include <stddef.h>
#include <stdbool.h>
#include <stdint.h>
#include <time.h>
```

Include dependency graph for library.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct obsws_config_t

  Connection configuration structure.
- struct obsws_stats_t

  Connection statistics - useful for monitoring and debugging connection health.
- struct obsws_response_t

  Response structure for requests to OBS.

Macros

- #define _POSIX_C_SOURCE 200809L

Typedefs

- typedef struct obsws_connection obsws_connection_t
- typedef void(∗ obsws_log_callback_t) (obsws_log_level_t level, const char ∗message, void ∗user_data)

  Log callback function type - called when the library generates log messages.
- typedef void(∗ obsws_event_callback_t) (obsws_connection_t ∗conn, const char ∗event_type, const char ∗event_data, void ∗user_data)
- typedef void(∗ obsws_state_callback_t) (obsws_connection_t ∗conn, obsws_state_t old_state, obsws_state_t new_state, void ∗user_data)

  State callback function type - called when connection state changes.

Enumerations

- enum obsws_error_t {
  OBSWS_OK = 0 , OBSWS_ERROR_INVALID_PARAM = -1 , OBSWS_ERROR_CONNECTION_FAILED = -2 , OBSWS_ERROR_SEND_FAILED = -5 ,
  OBSWS_ERROR_RECV_FAILED = -6 , OBSWS_ERROR_SSL_FAILED = -11 , OBSWS_ERROR_AUTH_FAILED = -3 , OBSWS_ERROR_PARSE_FAILED = -7 ,
  OBSWS_ERROR_NOT_CONNECTED = -8 , OBSWS_ERROR_ALREADY_CONNECTED = -9 , OBSWS_ERROR_TIMEOUT = -4 , OBSWS_ERROR_OUT_OF_MEMORY = -10 ,
  OBSWS_ERROR_UNKNOWN = -99 }

Error codes returned by library functions.

- enum obsws_state_t { OBSWS_STATE_DISCONNECTED = 0 , OBSWS_STATE_CONNECTING = 1 , OBSWS_STATE_AUTHENTICATING = 2 , OBSWS_STATE_CONNECTED = 3 , OBSWS_STATE_ERROR = 4 }

  Connection state - represents the current phase of the connection lifecycle.
- enum obsws_log_level_t { OBSWS_LOG_NONE = 0 , OBSWS_LOG_ERROR = 1 , OBSWS_LOG_WARNING = 2 , OBSWS_LOG_INFO = 3 , OBSWS_LOG_DEBUG = 4 }

  Log levels for filtering library output.
- enum obsws_debug_level_t { OBSWS_DEBUG_NONE = 0 , OBSWS_DEBUG_LOW = 1 , OBSWS_DEBUG_MEDIUM = 2 , OBSWS_DEBUG_HIGH = 3 }

  Debug levels - fine-grained control for troubleshooting connection issues.

Functions

- obsws_error_t obsws_init (void)

  Initialize the OBS WebSocket library.
- void obsws_cleanup (void)

  Cleanup the OBS WebSocket library.
- const char * obsws_version (void)

  Get the library version string.
- void obsws_set_log_level (obsws_log_level_t level)

  Set the global log level for the library.
- void obsws_set_debug_level (obsws_debug_level_t level)

  Set the global debug level for the library.
- obsws_debug_level_t obsws_get_debug_level (void)

  Get the current global debug level.
- void obsws_config_init (obsws_config_t *config)

  Create a default configuration structure with reasonable defaults.
- obsws_connection_t * obsws_connect (const obsws_config_t *config)

  Establish a connection to OBS.
- void obsws_disconnect (obsws_connection_t *conn)

  Disconnect and destroy a connection.
- bool obsws_is_connected (const obsws_connection_t *conn)

  Check if connection is currently authenticated and ready to use.
- obsws_state_t obsws_get_state (const obsws_connection_t *conn)

  Get the detailed current connection state.
- obsws_error_t obsws_get_stats (const obsws_connection_t *conn, obsws_stats_t *stats)

  Get connection statistics and performance metrics.
- obsws_error_t obsws_reconnect (obsws_connection_t *conn)

  Manually trigger a reconnection attempt.
- int obsws_ping (obsws_connection_t *conn, uint32_t timeout_ms)

  Send a ping and measure round-trip time to check connection health.
- obsws_error_t obsws_set_current_scene (obsws_connection_t *conn, const char *scene_name, obsws_response_t **response)

  Switch OBS to a specific scene.
- obsws_error_t obsws_get_current_scene (obsws_connection_t *conn, char *scene_name, size_t buffer_↩ size)

  Query the currently active scene in OBS.

- obsws_error_t obsws_get_scene_list (obsws_connection_t *conn, char ***scenes, size_t *count)
- obsws_error_t obsws_set_scene_collection (obsws_connection_t *conn, const char *collection_name, obsws_response_t **response)

  Switch to a different scene collection.
- obsws_error_t obsws_start_recording (obsws_connection_t *conn, obsws_response_t **response)

  Start recording in OBS.
- obsws_error_t obsws_stop_recording (obsws_connection_t *conn, obsws_response_t **response)

  Stop recording.
- obsws_error_t obsws_start_streaming (obsws_connection_t *conn, obsws_response_t **response)

  Start streaming in OBS.
- obsws_error_t obsws_stop_streaming (obsws_connection_t *conn, obsws_response_t **response)

  Stop streaming.
- obsws_error_t obsws_get_streaming_status (obsws_connection_t *conn, bool *is_streaming, obsws_response_t **response)
- obsws_error_t obsws_get_recording_status (obsws_connection_t *conn, bool *is_recording, obsws_response_t **response)

  Get whether OBS is currently recording.
- obsws_error_t obsws_set_source_visibility (obsws_connection_t *conn, const char *scene_name, const char *source_name, bool visible, obsws_response_t **response)
- obsws_error_t obsws_set_source_filter_enabled (obsws_connection_t *conn, const char *source_name, const char *filter_name, bool enabled, obsws_response_t **response)
- obsws_error_t obsws_send_request (obsws_connection_t *conn, const char *request_type, const char *request_data, obsws_response_t **response, uint32_t timeout_ms)

  Send a synchronous request to OBS and wait for the response.
- int obsws_process_events (obsws_connection_t *conn, uint32_t timeout_ms)

  Process pending events from the WebSocket connection.
- void obsws_response_free (obsws_response_t *response)

  Free a response structure and all its allocated memory.
- const char * obsws_error_string (obsws_error_t error)

  Convert an error code to a human-readable string.
- const char * obsws_state_string (obsws_state_t state)

  Convert a connection state to a human-readable string.
- void obsws_free_scene_list (char **scenes, size_t count)

4.3.1 Macro Definition Documentation

4.3.1.1 _POSIX_C_SOURCE

```
#define _POSIX_C_SOURCE 200809L
```

Definition at line 5 of file library.h.

4.3.2 Typedef Documentation

4.3.2.1 obsws_connection_t

```
typedef struct obsws_connection obsws_connection_t
```

Definition at line 142 of file library.h.

4.3.2.2 obsws_event_callback_t

```
typedef void(* obsws_event_callback_t) (obsws_connection_t *conn, const char *event_type, const
char *event_data, void *user_data)
```

Definition at line 190 of file library.h.

4.3.2.3 obsws_log_callback_t

```
typedef void(* obsws_log_callback_t) (obsws_log_level_t level, const char *message, void *user↩
_data)
```

Log callback function type - called when the library generates log messages.

This callback gives you a chance to handle logging however you want - write to a file, display in a GUI, send to a remote server, etc. If you don't provide a log callback, messages go to stderr by default.

Parameters

level	The severity level of this log message (error, warning, info, debug)
message	The actual log message text (null-terminated string)
user_data	Pointer you provided in the config - use for context

Note

The message buffer is temporary and may be freed after this callback returns. If you need to keep the message, copy it with strdup() or similar.

This callback is called from an internal thread, so if you access shared data structures, protect them with mutexes.

Avoid doing expensive operations in this callback - logging should be fast.

Definition at line 161 of file library.h.

4.3.2.4 obsws_state_callback_t

```
typedef void(* obsws_state_callback_t) (obsws_connection_t *conn, obsws_state_t old_state,
obsws_state_t new_state, void *user_data)
```

State callback function type - called when connection state changes.

This is how you know when the connection comes up or goes down. Use this to update your UI - disable buttons when disconnected, enable them when connected, show spinners during connecting, etc.

The callback receives both the old and new states so you can see the transition. For example, if old_state is DISCONNECTED and new_state is CONNECTING, you might show a "connecting..." message. If old_state is CONNECTED and new_state is DISCONNECTED, you might show "disconnected" and disable sending commands.

Parameters

conn	The connection whose state changed
old_state	Previous state (what it was before)
new_state	Current state (what it is now)
user_data	Pointer you provided in the config

Note

This callback is called from an internal thread, so protect shared data.

Don't do slow operations here - state changes should be handled quickly.

State transitions:

- DISCONNECTED -> CONNECTING (connection attempt starting)
- CONNECTING -> AUTHENTICATING (WebSocket connected, checking auth)
- AUTHENTICATING -> CONNECTED (ready to use)
- CONNECTING -> ERROR (connection failed)
- AUTHENTICATING -> ERROR (auth failed)
- CONNECTED -> DISCONNECTED (user disconnected or connection lost)
- CONNECTED -> ERROR (unexpected connection drop)
- ERROR -> DISCONNECTED (after cleanup)
- Any state -> DISCONNECTED (when you call obsws_disconnect)

Definition at line 223 of file library.h.

4.3.3 Enumeration Type Documentation

4.3.3.1 obsws_debug_level_t

```
enum obsws_debug_level_t
```

Debug levels - fine-grained control for troubleshooting connection issues.

Debug output is separate from log output because it's meant for developers debugging the library itself. It shows low-level protocol details. You probably only need this if something seems broken or you're curious about the protocol.

WARNING: Debug level HIGH will log passwords and raw messages. Never use in production or with untrusted users watching the output.

Enumerator

OBSWS_DEBUG_NONE	
OBSWS_DEBUG_LOW	
OBSWS_DEBUG_MEDIUM	
OBSWS_DEBUG_HIGH	

Definition at line 134 of file library.h.

```
00134             {
00135     OBSWS_DEBUG_NONE = 0,       /* No debug output - production mode */
00136     OBSWS_DEBUG_LOW = 1,        /* Connection events, auth success/failure, major state changes */
00137     OBSWS_DEBUG_MEDIUM = 2,     /* Low + WebSocket opcodes, event type names, request IDs */
00138     OBSWS_DEBUG_HIGH = 3        /* Medium + full message contents - can include passwords!  */
00139 } obsws_debug_level_t;
```

4.3.3.2 obsws_error_t

```
enum obsws_error_t
```

Error codes returned by library functions.

These error codes provide detailed information about what went wrong during library operations. Unlike generic error codes, they help distinguish between different failure modes so you can implement proper error handling and recovery strategies. For example, OBSWS_ERROR_TIMEOUT means you should probably retry the operation, while OBSWS_ERROR_AUTH_FAILED means retrying won't help - the password is just wrong.

The library uses negative error codes following POSIX conventions. Zero is always success. This makes it easy to check errors with simple conditions like if (error $<$ 0).

Note that some errors are recoverable (network timeouts, temporary connection failures) while others are not (invalid parameters, authentication failure). The auto-reconnect feature only applies to network-level errors, not application errors like wrong scene names.

Enumerator

OBSWS_OK	
OBSWS_ERROR_INVALID_PARAM	
OBSWS_ERROR_CONNECTION_FAILED	
OBSWS_ERROR_SEND_FAILED	
OBSWS_ERROR_RECV_FAILED	
OBSWS_ERROR_SSL_FAILED	
OBSWS_ERROR_AUTH_FAILED	
OBSWS_ERROR_PARSE_FAILED	
OBSWS_ERROR_NOT_CONNECTED	
OBSWS_ERROR_ALREADY_CONNECTED	
OBSWS_ERROR_TIMEOUT	
OBSWS_ERROR_OUT_OF_MEMORY	
OBSWS_ERROR_UNKNOWN	

Definition at line 51 of file library.h.

```
00051             {
00052     OBSWS_OK = 0,
```

```
00053
00054     /* Parameter validation errors (application layer) - not recoverable by retrying */
00055     OBSWS_ERROR_INVALID_PARAM = -1,
00056
00057     /* Network-level errors (can be recovered with reconnection) */
00058     OBSWS_ERROR_CONNECTION_FAILED = -2,
00059     OBSWS_ERROR_SEND_FAILED = -5,
00060     OBSWS_ERROR_RECV_FAILED = -6,
00061     OBSWS_ERROR_SSL_FAILED = -11,
00062
00063     /* Authentication errors (recoverable only by fixing the password) */
00064     OBSWS_ERROR_AUTH_FAILED = -3,
00065
00066     /* Protocol/messaging errors (typically indicate bad request data or OBS issues) */
00067     OBSWS_ERROR_PARSE_FAILED = -7,
00068     OBSWS_ERROR_NOT_CONNECTED = -8,
00069     OBSWS_ERROR_ALREADY_CONNECTED = -9,
00070
00071     /* Timeout errors (recoverable by retrying with patience) */
00072     OBSWS_ERROR_TIMEOUT = -4,
00073
00074     /* System resource errors (usually indicates system-wide issues) */
00075     OBSWS_ERROR_OUT_OF_MEMORY = -10,
00076
00077     /* Catch-all for things we didn't expect */
00078     OBSWS_ERROR_UNKNOWN = -99
00079 } obsws_error_t;
```

4.3.3.3 obsws_log_level_t

```
enum obsws_log_level_t
```

Log levels for filtering library output.

Think of log levels like a funnel - higher levels include all the output from lower levels plus more. So LOG_DEBUG includes everything, while LOG_ERROR only shows when things go wrong.

For production, use OBSWS_LOG_ERROR or OBSWS_LOG_WARNING to avoid spam. For development/debugging, use OBSWS_LOG_DEBUG to see everything happening.

Enumerator

OBSWS_LOG_NONE	
OBSWS_LOG_ERROR	
OBSWS_LOG_WARNING	
OBSWS_LOG_INFO	
OBSWS_LOG_DEBUG	

Definition at line 116 of file library.h.

```
00116             {
00117     OBSWS_LOG_NONE = 0,         /* Silence the library completely */
00118     OBSWS_LOG_ERROR = 1,        /* Only errors - something went wrong */
00119     OBSWS_LOG_WARNING = 2,      /* Errors + warnings - potential issues but still working */
00120     OBSWS_LOG_INFO = 3,         /* Normal operation info, good for seeing what's happening */
00121     OBSWS_LOG_DEBUG = 4         /* Very verbose, includes internal decisions and state changes */
00122 } obsws_log_level_t;
```

4.3.3.4 obsws_state_t

```
enum obsws_state_t
```

Connection state - represents the current phase of the connection lifecycle.

The connection goes through several states as it initializes. Understanding these states is important because different operations are only valid in certain states. For example, you can't send scene-switching commands when the state is CONNECTING

- you have to wait until CONNECTED.

The state machine looks like this: DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED Any state can transition to ERROR if something goes wrong CONNECTED or ERROR can go back to DISCONNECTED when closing

When you get a state callback, it tells you the old and new states so you can react appropriately. For example, you might want to disable UI buttons when moving from CONNECTED to DISCONNECTED.

Enumerator

OBSWS_STATE_DISCONNECTED	
OBSWS_STATE_CONNECTING	
OBSWS_STATE_AUTHENTICATING	
OBSWS_STATE_CONNECTED	
OBSWS_STATE_ERROR	

Definition at line 98 of file library.h.

```
00098             {
00099     OBSWS_STATE_DISCONNECTED = 0,       /* Not connected to OBS, no operations possible */
00100     OBSWS_STATE_CONNECTING = 1,         /* WebSocket handshake in progress, wait for AUTHENTICATING */
00101     OBSWS_STATE_AUTHENTICATING = 2,     /* Connected but still doing auth, wait for CONNECTED */
00102     OBSWS_STATE_CONNECTED = 3,          /* Ready - authentication complete, send commands now */
00103     OBSWS_STATE_ERROR = 4               /* Unrecoverable error occurred, reconnection might help */
00104 } obsws_state_t;
```

4.3.4 Function Documentation

4.3.4.1 obsws_cleanup()

```
void obsws_cleanup (
            void  )
```

Cleanup the OBS WebSocket library.

Call this when done using the library to release resources. Any connections should be disconnected before cleanup, though the library will try to clean them up if they're not. After calling this, don't use any library functions until you call obsws_init() again.

Note

It's safe to call this multiple times.

You should obsws_disconnect() all connections before calling this.

Threads should exit before calling cleanup - don't cleanup while callbacks are running.

Cleanup the OBS WebSocket library.

Call this when you're done with the library to deallocate OpenSSL resources. This is a counterpart to obsws_init().

Important: Make sure all obsws_connection_t objects have been disconnected and freed via obsws_disconnect() before calling this. If not, you might have dangling references and resource leaks.

Thread safety: This function is thread-safe and idempotent (safe to call multiple times). It checks g_library_initialized before doing anything.

Note: This is optional on program exit because the OS will clean up memory anyway. But it's good practice for:

- Library consumers that need clean shutdown
- Memory leak detectors / Valgrind tests
- Programs that unload the library

See also

obsws_init

Definition at line 1237 of file library.c.

```
01237                           {
01238     pthread_mutex_lock(&g_init_mutex);
01239
01240     if (!g_library_initialized) {
01241         pthread_mutex_unlock(&g_init_mutex);
01242         return;
01243     }
01244
01245     EVP_cleanup();
01246     g_library_initialized = false;
01247
01248     pthread_mutex_unlock(&g_init_mutex);
01249 }
```

4.3.4.2 obsws_config_init()

```
void obsws_config_init (
            obsws_config_t * config )
```

Create a default configuration structure with reasonable defaults.

This initializes a config structure with settings that should work for most cases. You then modify only the fields you care about. This is better than manually setting each field because if we add new config options in the future, you'll automatically get the right defaults without changing your code.

After calling this, typically you'd set: config.host = "192.168.1.100"; config.port = 4455; config.password = "your-obs-password";

Then pass it to obsws_connect().

Parameters

config	Pointer to configuration structure to fill with defaults

Create a default configuration structure with reasonable defaults.

Before calling obsws_connect(), you create an obsws_config_t structure with the connection parameters. This function initializes that structure with sensible defaults so you only need to change what's different for your use case.

Default values set:

- port: 4455 (OBS WebSocket v5 default port)
- use_ssl: false (OBS uses ws://, not wss://)
- connect_timeout_ms: 5000 (5 seconds to connect)
- recv_timeout_ms: 5000 (5 seconds to receive each message)
- send_timeout_ms: 5000 (5 seconds to send each message)
- ping_interval_ms: 10000 (send ping every 10 seconds)
- ping_timeout_ms: 5000 (expect pong within 5 seconds)
- auto_reconnect: true (reconnect automatically if connection drops)
- reconnect_delay_ms: 1000 (start with 1 second delay)
- max_reconnect_delay_ms: 30000 (max wait is 30 seconds)
- max_reconnect_attempts: 0 (infinite attempts)

After calling this, you typically set:

- config.host = "localhost" (where OBS is running)
- config.password = "your_password" (if OBS has auth enabled)
- config.event_callback = your_callback_func (to receive events)

Parameters

config	Pointer to structure to initialize (must not be NULL)

See also

obsws_connect

Definition at line 1367 of file library.c.

```
01367                                                {
01368     memset(config, 0, sizeof(obsws_config_t));
01369
01370     config->port = 4455;
01371     config->use_ssl = false;
01372     config->connect_timeout_ms = 5000;
01373     config->recv_timeout_ms = 5000;
```

```
01374     config->send_timeout_ms = 5000;
01375     config->ping_interval_ms = 10000;
01376     config->ping_timeout_ms = 5000;
01377     config->auto_reconnect = true;
01378     config->reconnect_delay_ms = 1000;
01379     config->max_reconnect_delay_ms = 30000;
01380     config->max_reconnect_attempts = 0; /* Infinite */
01381 }
```

4.3.4.3 obsws_connect()

```
obsws_connection_t * obsws_connect (
            const obsws_config_t * config )
```

Establish a connection to OBS.

This is the main entry point for using the library. You provide a configuration structure (initialized with obsws_config↩ _init and then customized), and this function connects to OBS, authenticates if needed, and spawns a background thread to handle incoming messages and events.

The function returns immediately - it doesn't wait for the connection to complete. Instead, it:

1. Creates a connection structure with the provided config
2. Allocates buffers for sending and receiving messages
3. Creates a libwebsockets context and connects to OBS
4. Spawns a background event_thread to process WebSocket messages
5. Returns the connection handle

Connection states: The connection progresses through states:

- DISCONNECTED -> CONNECTING (TCP handshake, WebSocket upgrade)
- CONNECTING -> AUTHENTICATING (receive HELLO, send IDENTIFY)
- AUTHENTICATING -> CONNECTED (receive IDENTIFIED)

You don't have to wait for CONNECTED state before calling obsws_send_request, but requests sent while not connected will return OBSWS_ERROR_NOT_CONNECTED.

Memory ownership: The connection structure is allocated and owned by this function. You must free it by calling obsws_disconnect(). Don't free it directly with free() - that will cause memory leaks (threads won't be cleaned up properly).

Error cases:

- NULL config or config->host: Returns NULL
- libwebsockets context creation fails: Returns NULL and logs error
- Network connection fails: Returns valid pointer but connection stays in ERROR state
- Bad password: Returns valid pointer but stays in AUTHENTICATING (never reaches CONNECTED)

Note: This function calls obsws_init() automatically if the library isn't already initialized.

Parameters

config	Pointer to initialized obsws_config_t with connection parameters

Returns

Pointer to new connection handle, or NULL if creation failed

See also

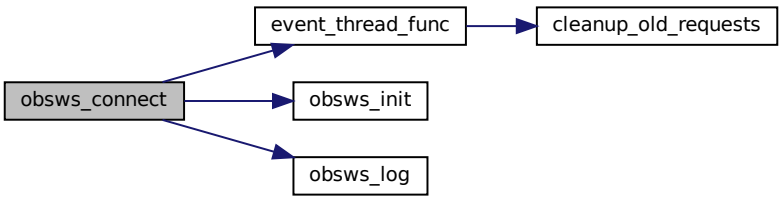
obsws_disconnect, obsws_get_state, obsws_send_request

Definition at line 1425 of file library.c.

```
01425                                                                   {
01426     if (!g_library_initialized) {
01427         obsws_init();
01428     }
01429 
01430     if (!config || !config->host) {
01431         return NULL;
01432     }
01433 
01434     obsws_connection_t *conn = calloc(1, sizeof(obsws_connection_t));
01435     if (!conn) return NULL;
01436 
01437     /* Copy configuration */
01438     memcpy(&conn->config, config, sizeof(obsws_config_t));
01439     if (config->host) conn->config.host = strdup(config->host);
01440     if (config->password) conn->config.password = strdup(config->password);
01441 
01442     /* Initialize mutexes */
01443     pthread_mutex_init(&conn->state_mutex, NULL);
01444     pthread_mutex_init(&conn->send_mutex, NULL);
01445     pthread_mutex_init(&conn->requests_mutex, NULL);
01446     pthread_mutex_init(&conn->stats_mutex, NULL);
01447     pthread_mutex_init(&conn->scene_mutex, NULL);
01448 
01449     /* Allocate buffers */
01450     conn->recv_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01451     conn->recv_buffer = malloc(conn->recv_buffer_size);
01452     conn->send_buffer_size = OBSWS_DEFAULT_BUFFER_SIZE;
01453     conn->send_buffer = malloc(conn->send_buffer_size);
01454 
01455     conn->state = OBSWS_STATE_DISCONNECTED;
01456     conn->current_reconnect_delay = config->reconnect_delay_ms;
01457 
01458     /* Create libwebsockets context */
01459     struct lws_context_creation_info info;
01460     memset(&info, 0, sizeof(info));
01461 
01462     info.port = CONTEXT_PORT_NO_LISTEN;
01463     info.protocols = protocols;
01464     info.gid = -1;
01465     info.uid = -1;
01466     info.options = LWS_SERVER_OPTION_DO_SSL_GLOBAL_INIT;
01467 
01468     conn->lws_context = lws_create_context(&info);
01469     if (!conn->lws_context) {
01470         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to create libwebsockets context");
01471         free(conn->recv_buffer);
01472         free(conn->send_buffer);
01473         free(conn);
01474         return NULL;
01475     }
01476 
01477     /* Connect to OBS */
01478     struct lws_client_connect_info ccinfo;
01479     memset(&ccinfo, 0, sizeof(ccinfo));
01480 
01481     ccinfo.context = conn->lws_context;
01482     ccinfo.address = conn->config.host;
01483     ccinfo.port = conn->config.port;
01484     ccinfo.path = "/";
01485     ccinfo.host = ccinfo.address;
01486     ccinfo.origin = ccinfo.address;
01487     ccinfo.protocol = protocols[0].name;
01488     ccinfo.userdata = conn;
```

```
01489
01490     if (config->use_ssl) {
01491         ccinfo.ssl_connection = LCCSCF_USE_SSL;
01492     }
01493
01494     conn->wsi = lws_client_connect_via_info(&ccinfo);
01495     if (!conn->wsi) {
01496         obsws_log(conn, OBSWS_LOG_ERROR, "Failed to initiate connection");
01497         lws_context_destroy(conn->lws_context);
01498         free(conn->recv_buffer);
01499         free(conn->send_buffer);
01500         free(conn);
01501         return NULL;
01502     }
01503
01504     /* Start event thread - protect flags with mutex */
01505     pthread_mutex_lock(&conn->state_mutex);
01506     conn->thread_running = true;
01507     conn->should_exit = false;
01508     pthread_mutex_unlock(&conn->state_mutex);
01509
01510     pthread_create(&conn->event_thread, NULL, event_thread_func, conn);
01511
01512     obsws_log(conn, OBSWS_LOG_INFO, "Connecting to OBS at %s:%d", config->host, config->port);
01513
01514     return conn;
01515 }
```

Here is the call graph for this function:



4.3.4.4 obsws_disconnect()

```
void obsws_disconnect (
            obsws_connection_t * conn )
```

Disconnect and destroy a connection.

This closes the connection to OBS and releases all associated resources. After calling this, the connection handle is invalid - don't use it anymore.

If auto_reconnect was enabled, it stops trying to reconnect. If you want to connect again, create a new connection with obsws_connect().

This function blocks until the background thread cleanly shuts down. It should be fast (under 100ms), but avoid calling it from callbacks since callbacks run in the connection thread - this would deadlock.

Parameters

conn	Connection handle to destroy (can be NULL, which does nothing)

Note

Safe to call even if already disconnected.

Don't call from inside callbacks (they run in the connection thread).

After calling this, all pending requests are abandoned (responses never arrive).

Disconnect and destroy a connection.

This is the counterpart to obsws_connect(). It cleanly shuts down the connection, stops the background event thread, and frees all allocated resources.

The function performs these steps:

1. Signal the event_thread to stop by setting should_exit flag
2. Wait for the event_thread to actually exit using pthread_join()
3. Send a normal WebSocket close frame to OBS (if connected)
4. Destroy the libwebsockets context
5. Free all pending requests (they won't get responses now, but don't leak memory)
6. Free buffers, config, authentication data
7. Destroy all mutexes and condition variables
8. Free the connection structure

After calling this, the connection pointer is invalid. Don't use it again.

Safe to call multiple times: If you call it twice, the second call will be a no-op (because conn will be NULL).

Safe to call even if connection never fully established: If you disconnect while in CONNECTING or AUTHENTICATING state, everything is still cleaned up.

Important: This function blocks until the event_thread exits. If you have a callback that's blocked, this will deadlock. Make sure your callbacks don't block!

Parameters

conn	The connection to close (can be NULL - safe to call)

See also

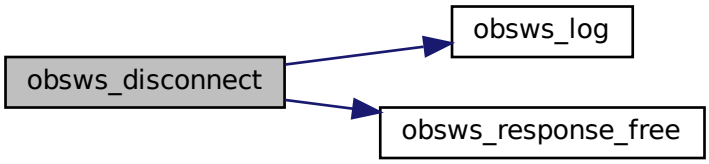
obsws_connect

Definition at line 1549 of file library.c.

```
01549                                                   {
01550     if (!conn) return;
01551
01552     obsws_log(conn, OBSWS_LOG_INFO, "Disconnecting from OBS");
01553
01554     /* Stop event thread - protect flag with mutex */
01555     pthread_mutex_lock(&conn->state_mutex);
01556     conn->should_exit = true;
01557     bool thread_was_running = conn->thread_running;
01558     pthread_mutex_unlock(&conn->state_mutex);
01559
01560     if (thread_was_running) {
```

```
01561         pthread_join(conn->event_thread, NULL);
01562     }
01563
01564     /* Close WebSocket - only if connected */
01565     if (conn->wsi && conn->state == OBSWS_STATE_CONNECTED) {
01566         lws_close_reason(conn->wsi, LWS_CLOSE_STATUS_NORMAL, NULL, 0);
01567     }
01568
01569     /* Cleanup libwebsockets */
01570     if (conn->lws_context) {
01571         lws_context_destroy(conn->lws_context);
01572     }
01573
01574     /* Free pending requests */
01575     pthread_mutex_lock(&conn->requests_mutex);
01576     pending_request_t *req = conn->pending_requests;
01577     while (req) {
01578         pending_request_t *next = req->next;
01579         if (req->response) {
01580             obsws_response_free(req->response);
01581         }
01582         pthread_mutex_destroy(&req->mutex);
01583         pthread_cond_destroy(&req->cond);
01584         free(req);
01585         req = next;
01586     }
01587     pthread_mutex_unlock(&conn->requests_mutex);
01588
01589     /* Free resources */
01590     free(conn->recv_buffer);
01591     free(conn->send_buffer);
01592     free((char *)conn->config.host);
01593     free((char *)conn->config.password);
01594     free(conn->challenge);
01595     free(conn->salt);
01596     free(conn->current_scene);
01597
01598     /* Destroy mutexes */
01599     pthread_mutex_destroy(&conn->state_mutex);
01600     pthread_mutex_destroy(&conn->send_mutex);
01601     pthread_mutex_destroy(&conn->requests_mutex);
01602     pthread_mutex_destroy(&conn->stats_mutex);
01603     pthread_mutex_destroy(&conn->scene_mutex);
01604
01605     free(conn);
01606 }
```

Here is the call graph for this function:



4.3.4.5 obsws_error_string()

```
const char * obsws_error_string (
            obsws_error_t error )
```

Convert an error code to a human-readable string.

Utility function for error reporting and logging. Returns a brief English description of each error code.

Never returns NULL Unknown error codes return "Unknown error", so it's always safe to use the returned pointer without NULL checks.

Example usage:

```
obsws_error_t err = obsws_send_request(...);
if (err != OBSWS_OK) {
    fprintf(stderr, "Error:  %s\\n", obsws_error_string(err));
}
```

Strings are constants Returned strings are statically allocated - do not modify or free them.

Parameters

error	The error code to describe

Returns

Pointer to a static string describing the error

See also

obsws_error_t

Definition at line 2229 of file library.c.

```
02229                                                   {
02230     switch (error) {
02231         case OBSWS_OK:  return "Success";
02232         case OBSWS_ERROR_INVALID_PARAM:  return "Invalid parameter";
02233         case OBSWS_ERROR_CONNECTION_FAILED:  return "Connection failed";
02234         case OBSWS_ERROR_AUTH_FAILED:  return "Authentication failed";
02235         case OBSWS_ERROR_TIMEOUT:  return "Timeout";
02236         case OBSWS_ERROR_SEND_FAILED:  return "Send failed";
02237         case OBSWS_ERROR_RECV_FAILED:  return "Receive failed";
02238         case OBSWS_ERROR_PARSE_FAILED:  return "Parse failed";
02239         case OBSWS_ERROR_NOT_CONNECTED:  return "Not connected";
02240         case OBSWS_ERROR_ALREADY_CONNECTED:  return "Already connected";
02241         case OBSWS_ERROR_OUT_OF_MEMORY:  return "Out of memory";
02242         case OBSWS_ERROR_SSL_FAILED:  return "SSL failed";
02243         default:  return "Unknown error";
02244     }
02245 }
```

4.3.4.6 obsws_free_scene_list()

```
void obsws_free_scene_list (
            char ** scenes,
            size_t count )
```

4.3.4.7 obsws_get_current_scene()

```
obsws_error_t obsws_get_current_scene (
            obsws_connection_t * conn,
            char * scene_name,
            size_t buffer_size )
```

Query the currently active scene in OBS.

This function queries OBS for the active scene name and returns it in the provided buffer. It also updates the local scene cache to keep it synchronized.

Cache Synchronization This function always queries the OBS server (doesn't use cached value). When the response arrives, it updates the cache. This ensures the library's cached scene name stays in sync with the actual OBS state.

Buffer Management The caller provides a buffer. If the scene name is longer than buffer_size-1, it will be truncated and null-terminated. Always check the returned buffer length if you need the full name.

Thread-safety The scene_mutex protects the cache update, so this is safe to call from any thread. Multiple concurrent calls are safe but will all query OBS (no deduplication).

Example usage:

```
char scene[256];
if (obsws_get_current_scene(conn, scene, sizeof(scene)) == OBSWS_OK) {
    printf("Current scene:  %s\\n", scene);
}
```

Parameters

conn	Connection object (must be in CONNECTED state)
scene_name	Buffer to receive the scene name (must not be NULL)
buffer_size	Size of scene_name buffer (must be > 0)

Returns

OBSWS_OK if scene name retrieved successfully

OBSWS_ERROR_INVALID_PARAM if conn, scene_name is NULL or buffer_size is 0

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

OBSWS_ERROR_PARSE_FAILED if response can't be parsed

See also

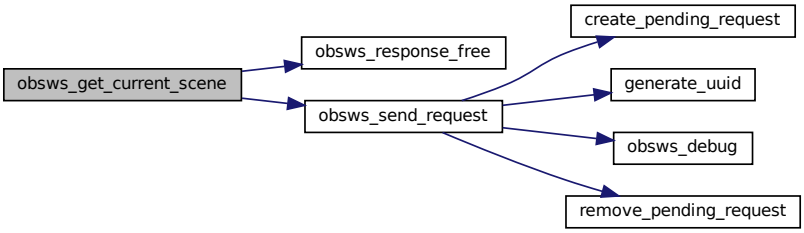
obsws_set_current_scene

Definition at line 2022 of file library.c.

```
02022
      {
02023     if (!conn || !scene_name || buffer_size == 0) {
02024         return OBSWS_ERROR_INVALID_PARAM;
02025     }
02026
02027     obsws_response_t *response = NULL;
02028     obsws_error_t result = obsws_send_request(conn, "GetCurrentProgramScene", NULL, &response, 0);
02029
02030     if (result == OBSWS_OK && response && response->success && response->response_data) {
02031         cJSON *data = cJSON_Parse(response->response_data);
```

```
02032         if (data) {
02033             cJSON *name = cJSON_GetObjectItem(data, "currentProgramSceneName");
02034             if (name && name->valuestring) {
02035                 strncpy(scene_name, name->valuestring, buffer_size - 1);
02036                 scene_name[buffer_size - 1] = '\0';
02037 
02038                 /* Update cache */
02039                 pthread_mutex_lock(&conn->scene_mutex);
02040                 free(conn->current_scene);
02041                 conn->current_scene = strdup(name->valuestring);
02042                 pthread_mutex_unlock(&conn->scene_mutex);
02043             }
02044             cJSON_Delete(data);
02045         }
02046     }
02047 
02048     if (response) {
02049         obsws_response_free(response);
02050     }
02051 
02052     return result;
02053 }
```

Here is the call graph for this function:



4.3.4.8 obsws_get_debug_level()

```
obsws_debug_level_t obsws_get_debug_level (
             void  )
```

Get the current global debug level.

Returns what the debug level is set to. Useful if you have code that needs to know how verbose the debugging is.

Returns

Current debug level

Get the current global debug level.

This is a read-only query - it doesn't change anything, just returns the current global debug level that was set by obsws_set_debug_level().

Useful for conditional logging in your application, e.g.:

```
if (obsws_get_debug_level() >= OBSWS_DEBUG_MEDIUM) {
    // do expensive trace operation
}
```

Returns

The currently active debug level

See also

obsws_set_debug_level

Definition at line 1333 of file library.c.

```
01333                                           {
01334     return g_debug_level;
01335 }
```

4.3.4.9 obsws_get_recording_status()

```
obsws_error_t obsws_get_recording_status (
            obsws_connection_t * conn,
            bool * is_recording,
            obsws_response_t ** response )
```

Get whether OBS is currently recording.

Returns true if recording is active, false otherwise. Similar to obsws_get_streaming_status() but for recording instead.

Parameters

conn	Connection handle (must be in CONNECTED state)
is_recording	Output parameter - true if recording, false if not
response	Optional pointer to receive full response with stats

Returns

OBSWS_OK on success

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if is_recording is NULL

Note

is_recording must be non-NULL, but response can be NULL.

4.3.4.10 obsws_get_scene_list()

```
obsws_error_t obsws_get_scene_list (
            obsws_connection_t * conn,
            char *** scenes,
            size_t * count )
```

4.3.4.11 obsws_get_state()

```
obsws_state_t obsws_get_state (
            const obsws_connection_t * conn )
```

Get the detailed current connection state.

Similar to obsws_is_connected() but returns the full state, not just a boolean. This lets you distinguish between different states - for example, you might show "connecting..." if the state is CONNECTING, vs "disconnected" if DISCONNECTED.

Parameters

conn	Connection handle

Returns

Current connection state (see obsws_state_t for the state machine)

Get the detailed current connection state.

Returns one of the connection states:

- OBSWS_STATE_DISCONNECTED: Not connected, idle
- OBSWS_STATE_CONNECTING: TCP connection established, waiting for WebSocket handshake
- OBSWS_STATE_AUTHENTICATING: WebSocket established, waiting for auth response
- OBSWS_STATE_CONNECTED: Connected and ready for requests
- OBSWS_STATE_ERROR: Connection encountered an error

State transitions normally follow this flow: DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED

But with errors, you can also have: (any state) -> ERROR

You don't usually need to call this - just try to send requests and check the error code. But it's useful for monitoring and debugging.

Thread-safe: This function locks the state_mutex, so it's safe to call from any thread.

Parameters

conn	The connection to check (NULL is safe - returns DISCONNECTED)

Returns

The current connection state

See also

obsws_is_connected

Definition at line 1663 of file library.c.

```
01663                                                                {
01664     if (!conn) return OBSWS_STATE_DISCONNECTED;
01665
01666     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
01667     obsws_state_t state = conn->state;
01668     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
01669
01670     return state;
01671 }
```

4.3.4.12 obsws_get_stats()

```
obsws_error_t obsws_get_stats (
            const obsws_connection_t * conn,
            obsws_stats_t * stats )
```

Get connection statistics and performance metrics.

Returns counters showing what's happened on this connection - messages sent, bytes transmitted, errors, reconnect attempts, etc. Useful for monitoring, debugging, or just curiosity.

The latency field (last_ping_ms) shows the round-trip time of the last ping, which indicates network quality. Higher values mean slower network or more congestion.

Parameters

conn	Connection handle
stats	Pointer to structure to fill (required)

Returns

OBSWS_OK on success, error code if conn is invalid

Note

fast operation - just copies the stats struct.

Get connection statistics and performance metrics.

The library maintains statistics about the connection:

- messages_sent / messages_received: Count of WebSocket messages
- bytes_sent / bytes_received: Total bytes transmitted/received
- connected_since: Timestamp of when we reached CONNECTED state
- reconnect_count: How many times we've reconnected (0 if never disconnected)

These can be useful for:

- Monitoring connection health
- Detecting stalled connections (if bytes_received stops increasing)
- Debugging and performance profiling
- Health dashboards or logging

Thread-safe: This function acquires stats_mutex and copies the entire stats structure, so it's safe to call from any thread. The copy operation is atomic from the caller's perspective.

Example usage:

```
obsws_stats_t stats;
obsws_get_stats(conn, &stats);
printf("Received %zu messages, %zu bytes\\n",
       stats.messages_received, stats.bytes_received);
```

Parameters

conn	The connection to query (NULL returns error)
stats	Pointer to stats structure to fill (must not be NULL)

Returns

OBSWS_OK on success, OBSWS_ERROR_INVALID_PARAM if conn or stats is NULL

See also

obsws_stats_t

Definition at line 1706 of file library.c.

```
01706                                                                                {
01707     if (!conn || !stats) return OBSWS_ERROR_INVALID_PARAM;
01708
01709     pthread_mutex_lock((pthread_mutex_t *)&conn->stats_mutex);
01710     memcpy(stats, &conn->stats, sizeof(obsws_stats_t));
01711     pthread_mutex_unlock((pthread_mutex_t *)&conn->stats_mutex);
01712
01713     return OBSWS_OK;
01714 }
```

4.3.4.13 obsws_get_streaming_status()

```
obsws_error_t obsws_get_streaming_status (
            obsws_connection_t * conn,
            bool * is_streaming,
            obsws_response_t ** response )
```

4.3.4.14 obsws_init()

```
obsws_error_t obsws_init (
            void  )
```

Initialize the OBS WebSocket library.

This must be called once, before using any other library functions. It sets up global state like threading primitives and initializes dependencies like libwebsockets and OpenSSL. If you call it multiple times, subsequent calls are ignored (thread-safe).

Typical usage: if (obsws_init() != OBSWS_OK) { fprintf(stderr, "Failed to init library\n"); return 1; } // ... use the library ... obsws_cleanup();

Returns

OBSWS_OK on success, error code if initialization failed

Note

Call this from your main thread before spawning other threads.

Very thread-safe - can be called multiple times, only initializes once.

Initialize the OBS WebSocket library.

This function must be called before creating any connections. It:

1. Initializes OpenSSL (EVP library for hashing)
2. Seeds the random number generator for UUID generation
3. Sets the global g_library_initialized flag

Thread safety: This function is thread-safe. Multiple threads can call it simultaneously, and only one will actually do the initialization (protected by g_init_mutex). Subsequent calls are no-ops.

Note: obsws_connect() will call this automatically if you forget, so you don't *have* to call it explicitly. But doing so allows you to initialize in a controlled way, separate from connection creation.

Cleanup: When you're done with the library, call obsws_cleanup() to deallocate resources. This is technically optional on program exit (the OS cleans up anyway), but good practice for testing and library shutdown.

Returns

OBSWS_OK always (initialization cannot fail in the current design)

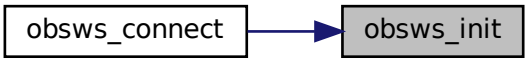
See also

obsws_cleanup, obsws_connect

Definition at line 1196 of file library.c.

```
01196                                {
01197     pthread_mutex_lock(&g_init_mutex);
01198 
01199     if (g_library_initialized) {
01200         pthread_mutex_unlock(&g_init_mutex);
01201         return OBSWS_OK;
01202     }
01203 
01204     /* Initialize OpenSSL */
01205     OpenSSL_add_all_algorithms();
01206 
01207     /* Seed random number generator */
01208     srand(time(NULL));
01209 
01210     g_library_initialized = true;
01211     pthread_mutex_unlock(&g_init_mutex);
01212 
01213     return OBSWS_OK;
01214 }
```

Here is the caller graph for this function:



4.3.4.15 obsws_is_connected()

```
bool obsws_is_connected (
            const obsws_connection_t * conn )
```

Check if connection is currently authenticated and ready to use.

Returns true only if the state is CONNECTED - meaning authentication completed and you can send commands. Returns false in all other states (DISCONNECTED, CONNECTING, AUTHENTICATING, ERROR).

This is the main function to check before sending commands. If it returns false, commands will fail with OBSWS_↩ ERROR_NOT_CONNECTED.

Parameters

conn	Connection handle

Returns

true if fully connected and authenticated, false otherwise

Note

Fast operation - just checks internal state variable.

Prefer using state callbacks to be notified of changes instead of polling.

Check if connection is currently authenticated and ready to use.

Convenience function that returns true if the connection is in CONNECTED state and false otherwise. Useful for checking before sending requests.

Thread-safe: This function locks the state_mutex before checking, so it's safe to call from any thread.

Return value: The connection must be in OBSWS_STATE_CONNECTED to return true. If it's CONNECTING, AUTHENTICATING, ERROR, or DISCONNECTED, this returns false.

Parameters

conn	The connection to check (NULL is safe - returns false)

Returns

true if connected, false otherwise

See also

obsws_get_state

Definition at line 1625 of file library.c.

```
01625                                                   {
01626     if (!conn) return false;
01627
01628     /* Thread-safe state check */
01629     pthread_mutex_lock((pthread_mutex_t *)&conn->state_mutex);
01630     bool connected = (conn->state == OBSWS_STATE_CONNECTED);
01631     pthread_mutex_unlock((pthread_mutex_t *)&conn->state_mutex);
01632
01633     return connected;
01634 }
```

4.3.4.16 obsws_ping()

```
int obsws_ping (
            obsws_connection_t * conn,
            uint32_t timeout_ms )
```

Send a ping and measure round-trip time to check connection health.

Sends a WebSocket ping to OBS and waits for the pong response. The round-trip time tells you the network latency. If the ping times out, it indicates a problem - either the network is very slow or OBS is not responding.

This is useful for:

- Checking if the connection is alive
- Measuring network latency
- Detecting when a connection appears OK but OBS isn't responding

Parameters

conn	Connection handle
timeout_ms	How long to wait for pong (milliseconds). Use 5000-10000 as typical.

Returns

Round-trip time in milliseconds if successful, negative error code if it failed (probably OBSWS_ERROR_↩ TIMEOUT if OBS isn't responding)

Note

Returns immediately if not connected - doesn't attempt to connect.

The library also sends pings automatically at ping_interval_ms, so you usually don't need to call this manually.

A high latency (e.g., several seconds) might indicate network problems.

4.3.4.17 obsws_process_events()

```
int obsws_process_events (
            obsws_connection_t * conn,
            uint32_t timeout_ms )
```

Process pending events from the WebSocket connection.

The library processes events in a background thread and calls your callbacks as events arrive. This function is provided for API compatibility and for applications that prefer to do event processing in the main loop rather than in background threads.

In most cases, you don't need to call this - just set up your callbacks and the library handles everything. Call this only if you want explicit control over when event processing happens (e.g., in a game loop).

Note: Even if you don't call this function, events are still processed in the background thread and callbacks are still called. This function is optional.

Parameters

conn	Connection handle
timeout_ms	Maximum time to wait for events (milliseconds). 0 = return immediately without processing, non-zero = wait up to this long for events.

Returns

Number of events processed, or negative error code

Note

This function is called automatically in the background thread, so you don't typically need to call it manually.

Callbacks are called from within this function (or from the background thread).

If you call this frequently with timeout_ms=0, you'll busy-wait (CPU usage).

This is provided mainly for API flexibility - most code should not use it.

Process pending events from the WebSocket connection.

This function is provided for API compatibility with single-threaded applications. However, the libwsv5 library uses a background event_thread by default, so this function is usually not needed - events are processed automatically in the background.

Background Event Processing: By design, all WebSocket messages (events, responses, etc.) are processed by the background event_thread. This thread:

- Continuously calls lws_service() to pump WebSocket events
- Receives incoming messages from OBS
- Routes responses to waiting callers via condition variables
- Calls event callbacks for real-time events
- Maintains keep-alive pings

Applications don't need to call this function - the thread handles everything.

What this function does: Currently, this is mainly for API compatibility. It:

- Validates the connection object
- If timeout_ms $>$ 0, sleeps for that duration
- Returns 0 (success)

When to use:

- Most applications: Don't call this - use background thread
- If you disable background thread: Call this in your main loop

Example (not typical - background thread is recommended):

```
// Not recommended - background thread is better
while (app_running) {
    obsws_process_events(conn, 100);  // Check every 100ms
}
```

Better approach - let background thread handle it:

```
obsws_connect(conn, "localhost", 4455, "password");
// Background thread processes events automatically
while (app_running) {
    // Your application code - no need to call process_events
}
obsws_disconnect(conn);
```

Parameters

conn	Connection object (can be NULL - returns error)
timeout_ms	Sleep duration in milliseconds (0 = don't sleep)

Returns

0 on success

OBSWS_ERROR_INVALID_PARAM if conn is NULL

See also

obsws_connect, obsws_disconnect, event_thread_func

Definition at line 2347 of file library.c.

```
02347                                                                   {
02348     if (!conn) return OBSWS_ERROR_INVALID_PARAM;
02349
02350     /* Events are processed in the background thread */
02351     /* This function is provided for API compatibility */
02352     if (timeout_ms > 0) {
02353         struct timespec ts;
02354         ts.tv_sec = timeout_ms / 1000;
02355         ts.tv_nsec = (timeout_ms % 1000) * 1000000;
02356         nanosleep(&ts, NULL);
02357     }
02358
02359     return 0;
02360 }
```

4.3.4.18 obsws_reconnect()

```
obsws_error_t obsws_reconnect (
            obsws_connection_t * conn )
```

Manually trigger a reconnection attempt.

Normally the library handles reconnection automatically if auto_reconnect is enabled. This function lets you force a reconnect attempt right now, for example if you detect the connection seems dead even though the library hasn't noticed yet.

If the connection is currently connected, this disconnects and reconnects. If it's not connected, this starts a connection attempt.

Parameters

conn	Connection handle

Returns

OBSWS_OK if reconnection started, error code otherwise

Note

This is async - it returns immediately, reconnection happens in background.

If auto_reconnect is disabled, this still reconnects one time.

4.3.4.19 obsws_response_free()

```
void obsws_response_free (
            obsws_response_t * response )
```

Free a response structure and all its allocated memory.

When you get a response from functions like obsws_set_current_scene() or obsws_send_request(), you're responsible for freeing it when done. This function frees the response_data and error_message strings, plus the response struct itself.

Always call this when you're done with a response, or you leak memory. It's safe to call with NULL (does nothing).

Parameters

response	Response to free (can be NULL)

Note

Safe to call with NULL.

After calling this, don't access the response pointer anymore.

Don't call free() manually on responses - use this function.

Free a response structure and all its allocated memory.

This function safely deallocates all memory associated with a response:

- The error_message string (if present)
- The response_data JSON string (if present)
- The response structure itself

Safe to call with NULL Calling with NULL is safe and does nothing - no crash or error. This allows for simpler cleanup:

```
obsws_response_t *resp = NULL;
obsws_send_request(..., &resp, ...);
obsws_response_free(resp);  // Safe even if send_request failed
```

Memory ownership

- obsws_send_request() allocates the response - you must free it
- High-level functions like obsws_set_current_scene() can optionally take response ownership or free internally based on parameters

NOT thread-safe Each response should only be accessed/freed by one thread. If multiple threads need the response, use higher-level synchronization.

Parameters

response	Response to free. Can be NULL (does nothing if so).

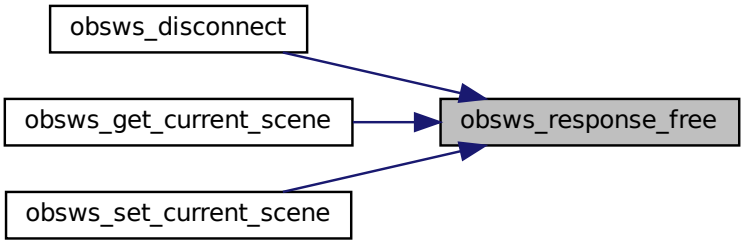
See also

obsws_send_request, obsws_response_t

Definition at line 2195 of file library.c.

```
02195                                                   {
02196     if (!response) return;
02197
02198     free(response->error_message);
02199     free(response->response_data);
02200     free(response);
02201 }
```

Here is the caller graph for this function:



4.3.4.20 obsws_send_request()

```
obsws_error_t obsws_send_request (
            obsws_connection_t * conn,
            const char * request_type,
            const char * request_data,
            obsws_response_t ** response,
            uint32_t timeout_ms )
```

Send a synchronous request to OBS and wait for the response.

This is the core function for all OBS operations. It implements the asynchronous request-response pattern of the OBS WebSocket v5 protocol:

Protocol Flow:

1. Generate a unique UUID for this request (used to match responses)
2. Create a pending_request_t to track the in-flight operation
3. Build the request JSON with opcode 6 (REQUEST)
4. Send the message via lws_write()
5. Block the caller with pthread_cond_timedwait() until response arrives
6. Return the response to caller (who owns it and must free with obsws_response_free)

Why synchronous from caller's perspective? Although WebSocket messages are async at the protocol level, we provide a synchronous API - the caller sends a request and blocks until the response arrives. This is simpler for application code than callback-based async APIs.

Behind the scenes, the background event_thread continuously processes WebSocket messages. When a REQUEST_RESPONSE (opcode 7) arrives matching a pending request ID, it signals the waiting condition variable, waking up the blocked caller.

Performance implications:

- Thread-safe: The main app thread can be blocked in obsws_send_request() while the background event_↩ thread processes other messages
- No polling: Uses condition variables, not CPU-wasting polling loops
- Can make multiple simultaneous requests from different threads (up to OBSWS_MAX_PENDING_↩ REQUESTS = 256)

Example usage:

```
obsws_response_t *response = NULL;
obsws_error_t err = obsws_send_request(conn, "SetCurrentProgramScene",
                                       "{\"sceneName\":  \"Scene1\"}",
                                       &response, 0);
if (err == OBSWS_OK && response && response->success) {
    printf("Scene switched successfully\\n");
}
obsws_response_free(response);
```

Parameters

conn	Connection object (must be in CONNECTED state)
request_type	OBS request type like "GetCurrentProgramScene", "SetCurrentProgramScene", etc.
request_data	Optional JSON string with request parameters. NULL for no parameters. Example: "{\"sceneName\": \"Scene1\"}"
response	Output pointer for the response. Will be allocated by this function. Caller must free with obsws_response_free(). Can be NULL if caller doesn't need the response (but response is still consumed from server).
timeout_ms	Timeout in milliseconds (0 = use config->recv_timeout_ms, typically 30000ms)

Returns

OBSWS_OK if response received (check response->success for operation success)

OBSWS_ERROR_INVALID_PARAM if conn, request_type, or response pointer is NULL

OBSWS_ERROR_NOT_CONNECTED if connection is not in CONNECTED state

OBSWS_ERROR_OUT_OF_MEMORY if pending request allocation fails

OBSWS_ERROR_SEND_FAILED if message send fails (buffer too small, invalid wsi, etc)

OBSWS_ERROR_TIMEOUT if no response received within timeout_ms

See also

obsws_response_t, obsws_response_free, obsws_error_string

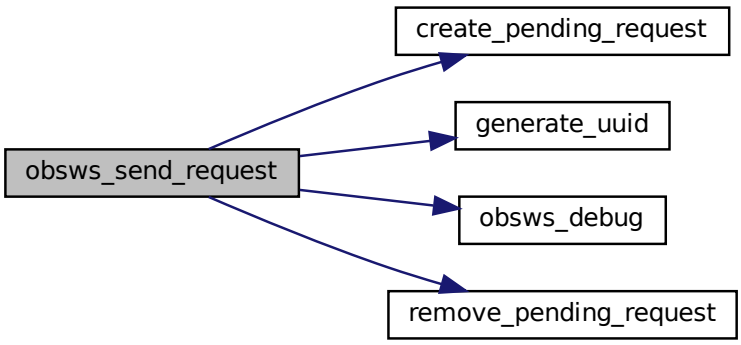
Definition at line 1776 of file library.c.

```
01777
      {
01778     if (!conn || !request_type || !response) {
```

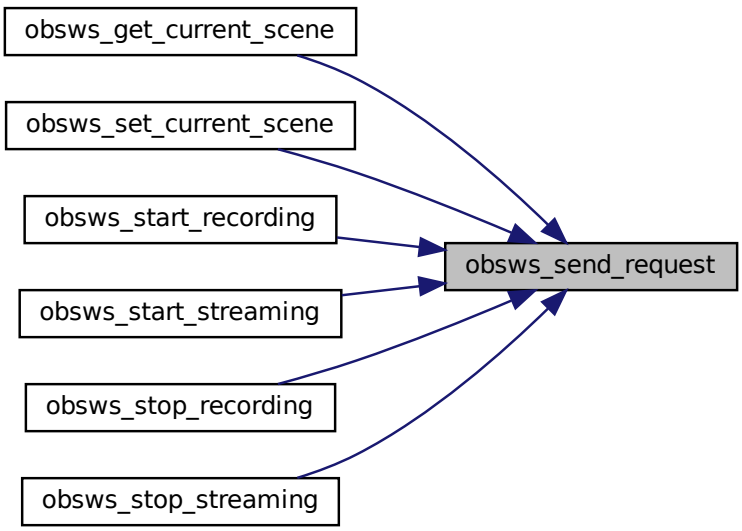
```
01779         return OBSWS_ERROR_INVALID_PARAM;
01780     }
01781
01782     if (conn->state != OBSWS_STATE_CONNECTED) {
01783         return OBSWS_ERROR_NOT_CONNECTED;
01784     }
01785
01786     /* Generate request ID */
01787     char request_id[OBSWS_UUID_LENGTH];
01788     generate_uuid(request_id);
01789
01790     /* Create pending request */
01791     pending_request_t *req = create_pending_request(conn, request_id);
01792     if (!req) {
01793         return OBSWS_ERROR_OUT_OF_MEMORY;
01794     }
01795
01796     /* Build request JSON */
01797     cJSON *request = cJSON_CreateObject();
01798     cJSON_AddNumberToObject(request, "op", OBSWS_OPCODE_REQUEST);
01799
01800     cJSON *d = cJSON_CreateObject();
01801     cJSON_AddStringToObject(d, "requestType", request_type);
01802     cJSON_AddStringToObject(d, "requestId", request_id);
01803
01804     if (request_data) {
01805         cJSON *data = cJSON_Parse(request_data);
01806         if (data) {
01807             cJSON_AddItemToObject(d, "requestData", data);
01808         }
01809     }
01810
01811     cJSON_AddItemToObject(request, "d", d);
01812
01813     char *message = cJSON_PrintUnformatted(request);
01814     cJSON_Delete(request);
01815
01816     /* DEBUG_HIGH: Show request being sent */
01817     obsws_debug(conn, OBSWS_DEBUG_HIGH, "Sending request (ID: %s):  %s", request_id, message);
01818
01819     /* Send request */
01820     pthread_mutex_lock(&conn->send_mutex);
01821     size_t len = strlen(message);
01822     obsws_error_t result = OBSWS_OK;
01823
01824     if (len < conn->send_buffer_size - LWS_PRE && conn->wsi) {
01825         memcpy(conn->send_buffer + LWS_PRE, message, len);
01826         int written = lws_write(conn->wsi, (unsigned char *)(conn->send_buffer + LWS_PRE), len,
      LWS_WRITE_TEXT);
01827
01828         if (written < 0) {
01829             result = OBSWS_ERROR_SEND_FAILED;
01830         } else {
01831             pthread_mutex_lock(&conn->stats_mutex);
01832             conn->stats.messages_sent++;
01833             conn->stats.bytes_sent += len;
01834             pthread_mutex_unlock(&conn->stats_mutex);
01835         }
01836     } else {
01837         result = OBSWS_ERROR_SEND_FAILED;
01838     }
01839     pthread_mutex_unlock(&conn->send_mutex);
01840
01841     free(message);
01842
01843     if (result != OBSWS_OK) {
01844         remove_pending_request(conn, req);
01845         return result;
01846     }
01847
01848     /* Wait for response */
01849     if (timeout_ms == 0) {
01850         timeout_ms = conn->config.recv_timeout_ms;
01851     }
01852
01853     struct timespec ts;
01854     clock_gettime(CLOCK_REALTIME, &ts);
01855     ts.tv_sec += timeout_ms / 1000;
01856     ts.tv_nsec += (timeout_ms % 1000) * 1000000;
01857     if (ts.tv_nsec >= 1000000000) {
01858         ts.tv_sec++;
01859         ts.tv_nsec -= 1000000000;
01860     }
01861
01862     pthread_mutex_lock(&req->mutex);
01863     while (!req->completed) {
01864         int wait_result = pthread_cond_timedwait(&req->cond, &req->mutex, &ts);
```

```
01865         if (wait_result == ETIMEDOUT) {
01866             pthread_mutex_unlock(&req->mutex);
01867             remove_pending_request(conn, req);
01868             return OBSWS_ERROR_TIMEOUT;
01869         }
01870     }
01871
01872     *response = req->response;
01873     req->response = NULL; /* Transfer ownership */
01874     pthread_mutex_unlock(&req->mutex);
01875
01876     remove_pending_request(conn, req);
01877
01878     return OBSWS_OK;
01879 }
```

Here is the call graph for this function:



Here is the caller graph for this function:



4.3.4.21 obsws_set_current_scene()

```
obsws_error_t obsws_set_current_scene (
            obsws_connection_t * conn,
            const char * scene_name,
            obsws_response_t ** response )
```

Switch OBS to a specific scene.

This is a high-level convenience function for scene switching. It demonstrates several important design patterns in the library:

Optimization: Scene Cache Before sending a request to OBS, this function checks the cached current scene. If the requested scene is already active, it returns immediately without network overhead. The cache is maintained by the event thread when SceneChanged events arrive.

Memory Ownership If the caller provides response pointer, they receive ownership and must call obsws_response_free(). If response is NULL, the function frees the response internally. This flexibility allows three usage patterns:

1. Check response: `obsws_set_current_scene(conn, name, &resp); if (resp->success) ...`
2. Ignore response: `obsws_set_current_scene(conn, name, NULL);` (response is freed internally)
3. Just check error: `if (obsws_send_request(...) != OBSWS_OK) ...`

Example usage:

```
// Pattern 1: Check response details
obsws_response_t *response = NULL;
if (obsws_set_current_scene(conn, "Scene1", &response) == OBSWS_OK &&
    response && response->success) {
    printf("Switched successfully\\n");
} else {
    printf("Switch failed: %s\\n", response ? response->error_message : "unknown");
}
obsws_response_free(response);
// Pattern 2: Ignore response (simpler)
obsws_set_current_scene(conn, "Scene1", NULL);
```

Thread-safety:

- Scene cache is protected by scene_mutex
- Safe to call from any thread
- Multiple calls can happen simultaneously (each uses send_mutex)

Parameters

conn	Connection object (must be in CONNECTED state)
scene_name	Name of the scene to switch to. Must not be NULL.
response	Optional output for response details. If provided, caller owns it and must free with obsws_response_free(). If NULL, response is freed internally.

Returns

OBSWS_OK if request sent and response received (check response->success for whether the scene switch actually succeeded in OBS)

OBSWS_ERROR_INVALID_PARAM if conn or scene_name is NULL

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if no response from OBS

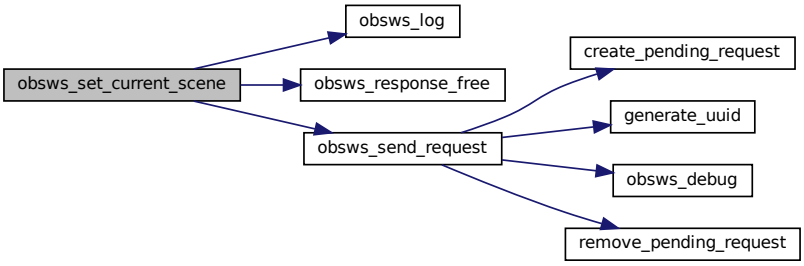
See also

obsws_get_current_scene, obsws_response_t, obsws_response_free

Definition at line 1935 of file library.c.

```
01935
      {
01936     if (!conn || !scene_name) {
01937         return OBSWS_ERROR_INVALID_PARAM;
01938     }
01939
01940     /* Check cache to avoid redundant switches */
01941     pthread_mutex_lock(&conn->scene_mutex);
01942     bool already_current = (conn->current_scene && strcmp(conn->current_scene, scene_name) == 0);
01943     pthread_mutex_unlock(&conn->scene_mutex);
01944
01945     if (already_current) {
01946         obsws_log(conn, OBSWS_LOG_DEBUG, "Already on scene:  %s", scene_name);
01947         if (response) {
01948             *response = calloc(1, sizeof(obsws_response_t));
01949             (*response)->success = true;
01950         }
01951         return OBSWS_OK;
01952     }
01953
01954     cJSON *request_data = cJSON_CreateObject();
01955     cJSON_AddStringToObject(request_data, "sceneName", scene_name);
01956
01957     char *data_str = cJSON_PrintUnformatted(request_data);
01958     cJSON_Delete(request_data);
01959
01960     obsws_response_t *resp = NULL;
01961     obsws_error_t result = obsws_send_request(conn, "SetCurrentProgramScene", data_str, &resp, 0);
01962     free(data_str);
01963
01964     if (result == OBSWS_OK && resp && resp->success) {
01965         pthread_mutex_lock(&conn->scene_mutex);
01966         free(conn->current_scene);
01967         conn->current_scene = strdup(scene_name);
01968         pthread_mutex_unlock(&conn->scene_mutex);
01969
01970         obsws_log(conn, OBSWS_LOG_INFO, "Switched to scene:  %s", scene_name);
01971     }
01972
01973     if (response) {
01974         *response = resp;
01975     } else if (resp) {
01976         obsws_response_free(resp);
01977     }
01978
01979     return result;
01980 }
```

Here is the call graph for this function:



4.3.4.22 obsws_set_debug_level()

```
void obsws_set_debug_level (
            obsws_debug_level_t level )
```

Set the global debug level for the library.

This controls detailed internal logging separate from regular log messages. Use this for troubleshooting connection/protocol issues. Each level includes all output from lower levels.

Debug output shows protocol-level information - opcodes, message IDs, etc. It's verbose and should only be used during development.

WARNING: Level HIGH logs passwords and raw messages. Never use in production or with untrusted users watching the output.

Parameters

level	Debug verbosity level

Set the global debug level for the library.

Debug logging is separate from regular logging. It provides extremely detailed trace information about the WebSocket protocol, message parsing, authentication, etc. This is useful during development and troubleshooting.

Debug levels:

- OBSWS_DEBUG_NONE: No debug output (fastest)
- OBSWS_DEBUG_LOW: Major state transitions and connection events
- OBSWS_DEBUG_MEDIUM: Message types and handlers invoked
- OBSWS_DEBUG_HIGH: Full message content and every operation

Debug logging is independent of log level. You can have OBSWS_LOG_ERROR set (hide non-error logs) but still see OBSWS_DEBUG_HIGH output.

Performance warning: OBSWS_DEBUG_HIGH produces enormous output and will slow down the library significantly. Only use during debugging!

Thread safety: Same as obsws_set_log_level (modifies global without locking).

Parameters

level	The debug verbosity level

See also

obsws_set_log_level, obsws_get_debug_level

Definition at line 1313 of file library.c.

```
01313                                                   {
01314     g_debug_level = level;
01315 }
```

4.3.4.23 obsws_set_log_level()

```
void obsws_set_log_level (
            obsws_log_level_t level )
```

Set the global log level for the library.

This affects all library output - messages at this level and below are shown. For example, if you set OBSWS_LOG↩_WARNING, you'll see warnings and errors but not info or debug messages. This setting affects all connections.

Common usage:

- Development: OBSWS_LOG_DEBUG - see everything while developing
- Testing: OBSWS_LOG_INFO - see what's happening
- Production: OBSWS_LOG_ERROR or OBSWS_LOG_WARNING - only see problems

Parameters

level	Minimum log level to output

Set the global log level for the library.

The library logs various messages during operation. This function sets which messages are displayed. All messages at the specified level and higher severity are shown; lower severity messages are hidden.

Levels in increasing severity:

- OBSWS_LOG_DEBUG: Low-level diagnostic info (too verbose for production)

- OBSWS_LOG_INFO: General informational messages (usual choice)
- OBSWS_LOG_WARNING: Potentially problematic situations (degraded but working)
- OBSWS_LOG_ERROR: Error conditions that need attention

Example: If you call obsws_set_log_level(OBSWS_LOG_WARNING), you'll see only WARNING and ERROR messages, but not INFO or DEBUG messages.

Thread safety: This modifies a global variable without locking, so if you might call this from multiple threads, use synchronization externally.

Parameters

level	The minimum severity level to display

See also

obsws_set_debug_level

Definition at line 1285 of file library.c.

```
01285                                                   {
01286     g_log_level = level;
01287 }
```

4.3.4.24 obsws_set_scene_collection()

```
obsws_error_t obsws_set_scene_collection (
            obsws_connection_t * conn,
            const char * collection_name,
            obsws_response_t ** response )
```

Switch to a different scene collection.

Scene collections are different sets of scenes. OBS can have multiple scene collections and you can switch between them. When you switch collections, all the scenes in the current collection are replaced with the scenes from the new collection. This is useful for different contexts (e.g., "Gaming", "IRL", "Voiceover", etc.).

Switching collections can take a moment because OBS needs to load all the new scenes and their settings. You'll get a SceneCollectionChanged event when done.

Parameters

conn	Connection handle (must be in CONNECTED state)
collection_name	Name of the scene collection to activate (case-sensitive, must exist in OBS)
response	Optional pointer to receive response (NULL to ignore)

Returns

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if collection_name is NULL

Note

Scene collection names are shown in OBS: Scene Collection dropdown.

Switching collections is slower than switching scenes (requires loading files).

This is async - function returns before OBS finishes switching.

If the collection doesn't exist, OBS returns an error in the response.

4.3.4.25 obsws_set_source_filter_enabled()

```
obsws_error_t obsws_set_source_filter_enabled (
            obsws_connection_t * conn,
            const char * source_name,
            const char * filter_name,
            bool enabled,
            obsws_response_t ** response )
```

4.3.4.26 obsws_set_source_visibility()

```
obsws_error_t obsws_set_source_visibility (
            obsws_connection_t * conn,
            const char * scene_name,
            const char * source_name,
            bool visible,
            obsws_response_t ** response )
```

4.3.4.27 obsws_start_recording()

```
obsws_error_t obsws_start_recording (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Start recording in OBS.

Tells OBS to begin recording the current scene composition to disk. The recording path and format are determined by OBS settings, not by this library.

This is a convenience wrapper around obsws_send_request() using the OBS "StartRecord" request type.

Return value interpretation:

- OBSWS_OK: Request was sent and OBS responded (check response->success)
- Other errors: Network/connection problem

Example usage:

```
obsws_response_t *resp = NULL;
if (obsws_start_recording(conn, &resp) == OBSWS_OK && resp && resp->success) {
    printf("Recording started\\n");
}
obsws_response_free(resp);
```

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

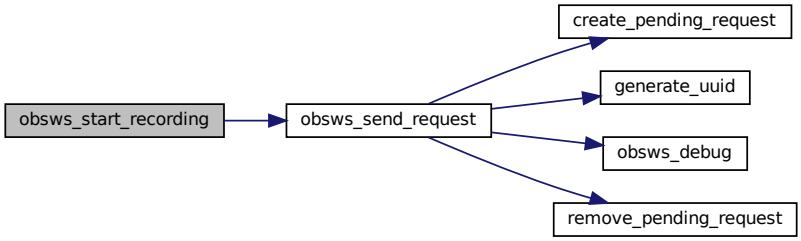
See also

obsws_stop_recording, obsws_send_request, obsws_response_free

Definition at line 2086 of file library.c.

```
02086                                                                                {
02087     return obsws_send_request(conn, "StartRecord", NULL, response, 0);
02088 }
```

Here is the call graph for this function:



4.3.4.28 obsws_start_streaming()

```
obsws_error_t obsws_start_streaming (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Start streaming in OBS.

Tells OBS to begin streaming to the configured destination (Twitch, YouTube, etc). The stream settings (URL, key, bitrate, etc) are determined by OBS settings.

This is a convenience wrapper around obsws_send_request() using the OBS "StartStream" request type.

Thread-safe: Safe to call from any thread while connected.

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

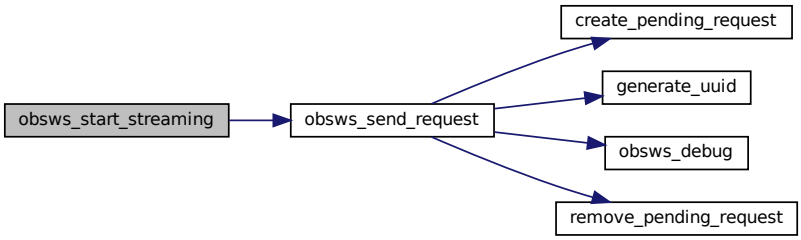
See also

obsws_stop_streaming, obsws_send_request, obsws_response_free

Definition at line 2139 of file library.c.

```
02139                                                                                {
02140     return obsws_send_request(conn, "StartStream", NULL, response, 0);
02141 }
```

Here is the call graph for this function:



4.3.4.29 obsws_state_string()

```
const char * obsws_state_string (
            obsws_state_t state )
```

Convert a connection state to a human-readable string.

Utility function for logging and debugging. Returns a brief English description of each connection state.

State Transitions:

- DISCONNECTED: Initial state or after disconnect()
- CONNECTING: connect() called, establishing TCP connection
- AUTHENTICATING: TCP connected, performing challenge-response auth

- CONNECTED: Auth succeeded, ready for requests
- ERROR: Network error or protocol failure, should reconnect

Valid transitions:

```
DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED
CONNECTED -> DISCONNECTED (on explicit disconnect)
CONNECTED -> ERROR (on network failure)
ERROR -> CONNECTING (on reconnect attempt)
```

Example usage:

```
obsws_state_t state = obsws_get_state(conn);
printf("Connection state:  %s\\n", obsws_state_string(state));
```

Never returns NULL - unknown states return "Unknown". Returned strings are static - do not modify or free.

Parameters

state	The connection state to describe

Returns

Pointer to a static string describing the state

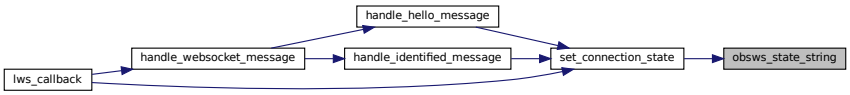
See also

obsws_get_state, obsws_state_t, obsws_is_connected

Definition at line 2282 of file library.c.

```
02282                                                {
02283     switch (state) {
02284         case OBSWS_STATE_DISCONNECTED:  return "Disconnected";
02285         case OBSWS_STATE_CONNECTING:  return "Connecting";
02286         case OBSWS_STATE_AUTHENTICATING:  return "Authenticating";
02287         case OBSWS_STATE_CONNECTED:  return "Connected";
02288         case OBSWS_STATE_ERROR:  return "Error";
02289         default:  return "Unknown";
02290     }
02291 }
```

Here is the caller graph for this function:



4.3.4.30 obsws_stop_recording()

```
obsws_error_t obsws_stop_recording (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Stop recording.

Tells OBS to stop recording and save the file. The saved recording will include everything from when you called obsws_start_recording() until now. If recording wasn't running, OBS ignores this (no error).

After calling this, OBS takes a moment to finalize the file (write headers, etc.), then sends a RecordingStateChanged event when complete. You can't immediately open the file - wait for the event first.

Parameters

conn	Connection handle (must be in CONNECTED state)
response	Optional pointer to receive response

Returns

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

Note

This is async - the file finalization happens after the function returns.

OBS sends a RecordingStateChanged event when the file is ready.

Don't try to move/open the file immediately - wait for the event first.

Stop recording.

Tells OBS to stop the currently active recording. If no recording is in progress, OBS returns success anyway (idempotent operation).

This is a convenience wrapper around obsws_send_request() using the OBS "StopRecord" request type.

Example usage:

```
if (obsws_stop_recording(conn, NULL) != OBSWS_OK) {
    fprintf(stderr, "Failed to stop recording\\n");
}
```

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

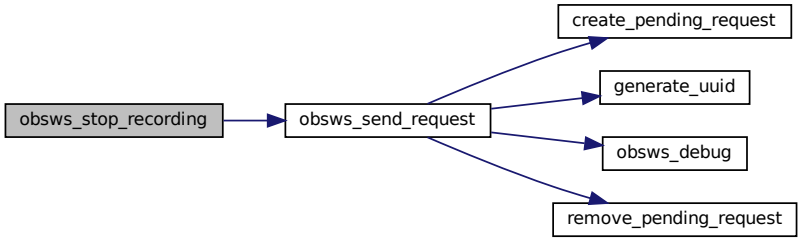
See also

obsws_start_recording, obsws_send_request, obsws_response_free

Definition at line 2115 of file library.c.

```
02115                                                                                {
02116     return obsws_send_request(conn, "StopRecord", NULL, response, 0);
02117 }
```

Here is the call graph for this function:



4.3.4.31 obsws_stop_streaming()

```
obsws_error_t obsws_stop_streaming (
            obsws_connection_t * conn,
            obsws_response_t ** response )
```

Stop streaming.

Tells OBS to stop streaming to the remote server. After calling this, OBS disconnects from the streaming server and the stream ends. The stopped event will tell you if the disconnect was clean or if there was an issue.

Parameters

conn	Connection handle (must be in CONNECTED state)
response	Optional pointer to receive response

Returns

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

Note

This is async - disconnecting from the server takes a moment.

OBS sends a StreamStateChanged event when streaming fully stops.

If not streaming, OBS ignores this request (no error).

Stop streaming.

Tells OBS to stop the active stream. If not currently streaming, OBS returns success anyway (idempotent operation).

This is a convenience wrapper around obsws_send_request() using the OBS "StopStream" request type.

Parameters

conn	Connection object (must be in CONNECTED state)
response	Optional output for response. Caller owns if provided, must free.

Returns

OBSWS_OK if response received (check response->success for success)

OBSWS_ERROR_NOT_CONNECTED if connection not ready

OBSWS_ERROR_TIMEOUT if OBS doesn't respond

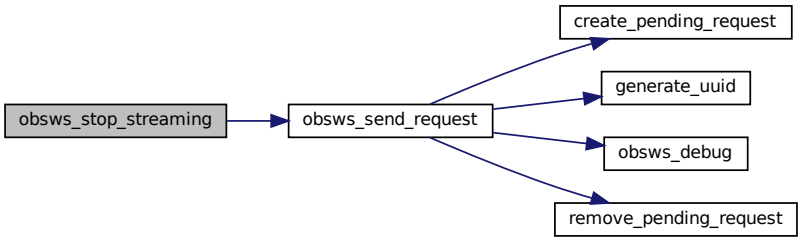
See also

obsws_start_streaming, obsws_send_request, obsws_response_free

Definition at line 2161 of file library.c.

```
02161                                                                                {
02162     return obsws_send_request(conn, "StopStream", NULL, response, 0);
02163 }
```

Here is the call graph for this function:



4.3.4.32 obsws_version()

```
const char * obsws_version (
            void  )
```

Get the library version string.

Returns a string like "1.0.0" indicating what version of the library you're using. Useful for debugging - you can log this at startup so you know which version of the library handled a particular issue.

Returns

Pointer to version string (valid until obsws_cleanup is called)

Returns a semantic version string like "1.0.0" that identifies which version of libwsv5 is being used. Useful for debugging and logging.

Returns

Pointer to static version string (don't free)

Definition at line 1259 of file library.c.

```
01259                                {
01260     return OBSWS_VERSION;
01261 }
```

4.4 library.h

Go to the documentation of this file.

```
00001 #ifndef LIBWSV5_LIBRARY_H
00002 #define LIBWSV5_LIBRARY_H
00003
00004 #ifndef _POSIX_C_SOURCE
00005 #define _POSIX_C_SOURCE 200809L
00006 #endif
00007
00008 #include <stddef.h>
00009 #include <stdbool.h>
00010 #include <stdint.h>
00011 #include <time.h>
00012
00013 #ifdef __cplusplus
00014 extern "C" {
00015 #endif
00016
00017 /* ============================================================================
00018 * OBS WebSocket v5 Protocol Library
00019 * ============================================================================
00020 * A robust and efficient implementation of the OBS WebSocket v5 protocol
00021 * for managing multiple OBS connections in multi-stream setups.
00022 *
00023 * Features:
00024 * - Connection management with automatic reconnection
00025 * - OBS WebSocket v5 authentication
00026 * - Scene switching and control
00027 * - Connection health monitoring
00028 * - Thread-safe operations
00029 * - Multiple concurrent connections support
00030 * ============================================================================
00031 */
00032
00033 /**
00034 * Error codes returned by library functions.
00035 *
00036 * These error codes provide detailed information about what went wrong during
00037 * library operations.  Unlike generic error codes, they help distinguish between
00038 * different failure modes so you can implement proper error handling and recovery
00039 * strategies.  For example, OBSWS_ERROR_TIMEOUT means you should probably retry
00040 * the operation, while OBSWS_ERROR_AUTH_FAILED means retrying won't help - the
00041 * password is just wrong.
00042 *
00043 * The library uses negative error codes following POSIX conventions.  Zero is always
00044 * success.  This makes it easy to check errors with simple conditions like if (error < 0).
00045 *
00046 * Note that some errors are recoverable (network timeouts, temporary connection
00047 * failures) while others are not (invalid parameters, authentication failure).
00048 * The auto-reconnect feature only applies to network-level errors, not application
00049 * errors like wrong scene names.
00050 */
00051 typedef enum {
00052     OBSWS_OK = 0,
00053
00054     /* Parameter validation errors (application layer) - not recoverable by retrying */
00055     OBSWS_ERROR_INVALID_PARAM = -1,
00056
00057     /* Network-level errors (can be recovered with reconnection) */
00058     OBSWS_ERROR_CONNECTION_FAILED = -2,
00059     OBSWS_ERROR_SEND_FAILED = -5,
00060     OBSWS_ERROR_RECV_FAILED = -6,
00061     OBSWS_ERROR_SSL_FAILED = -11,
00062
00063     /* Authentication errors (recoverable only by fixing the password) */
00064     OBSWS_ERROR_AUTH_FAILED = -3,
00065
00066     /* Protocol/messaging errors (typically indicate bad request data or OBS issues) */
00067     OBSWS_ERROR_PARSE_FAILED = -7,
00068     OBSWS_ERROR_NOT_CONNECTED = -8,
00069     OBSWS_ERROR_ALREADY_CONNECTED = -9,
00070
00071     /* Timeout errors (recoverable by retrying with patience) */
00072     OBSWS_ERROR_TIMEOUT = -4,
00073
00074     /* System resource errors (usually indicates system-wide issues) */
00075     OBSWS_ERROR_OUT_OF_MEMORY = -10,
00076
00077     /* Catch-all for things we didn't expect */
00078     OBSWS_ERROR_UNKNOWN = -99
00079 } obsws_error_t;
00080
00081 /**
00082 * Connection state - represents the current phase of the connection lifecycle.
```

```
00083 *
00084 * The connection goes through several states as it initializes.  Understanding these
00085 * states is important because different operations are only valid in certain states.
00086 * For example, you can't send scene-switching commands when the state is CONNECTING
00087 * - you have to wait until CONNECTED.
00088 *
00089 * The state machine looks like this:
00090 *   DISCONNECTED -> CONNECTING -> AUTHENTICATING -> CONNECTED
00091 *   Any state can transition to ERROR if something goes wrong
00092 *   CONNECTED or ERROR can go back to DISCONNECTED when closing
00093 *
00094 * When you get a state callback, it tells you the old and new states so you can
00095 * react appropriately.  For example, you might want to disable UI buttons when
00096 * moving from CONNECTED to DISCONNECTED.
00097 */
00098 typedef enum {
00099     OBSWS_STATE_DISCONNECTED = 0,       /* Not connected to OBS, no operations possible */
00100     OBSWS_STATE_CONNECTING = 1,         /* WebSocket handshake in progress, wait for AUTHENTICATING */
00101     OBSWS_STATE_AUTHENTICATING = 2,     /* Connected but still doing auth, wait for CONNECTED */
00102     OBSWS_STATE_CONNECTED = 3,          /* Ready - authentication complete, send commands now */
00103     OBSWS_STATE_ERROR = 4               /* Unrecoverable error occurred, reconnection might help */
00104 } obsws_state_t;
00105
00106 /**
00107 * Log levels for filtering library output.
00108 *
00109 * Think of log levels like a funnel - higher levels include all the output from
00110 * lower levels plus more.  So LOG_DEBUG includes everything, while LOG_ERROR only
00111 * shows when things go wrong.
00112 *
00113 * For production, use OBSWS_LOG_ERROR or OBSWS_LOG_WARNING to avoid spam.
00114 * For development/debugging, use OBSWS_LOG_DEBUG to see everything happening.
00115 */
00116 typedef enum {
00117     OBSWS_LOG_NONE = 0,         /* Silence the library completely */
00118     OBSWS_LOG_ERROR = 1,        /* Only errors - something went wrong */
00119     OBSWS_LOG_WARNING = 2,      /* Errors + warnings - potential issues but still working */
00120     OBSWS_LOG_INFO = 3,         /* Normal operation info, good for seeing what's happening */
00121     OBSWS_LOG_DEBUG = 4         /* Very verbose, includes internal decisions and state changes */
00122 } obsws_log_level_t;
00123
00124 /**
00125 * Debug levels - fine-grained control for troubleshooting connection issues.
00126 *
00127 * Debug output is separate from log output because it's meant for developers
00128 * debugging the library itself.  It shows low-level protocol details.  You probably
00129 * only need this if something seems broken or you're curious about the protocol.
00130 *
00131 * WARNING: Debug level HIGH will log passwords and raw messages.  Never use in
00132 * production or with untrusted users watching the output.
00133 */
00134 typedef enum {
00135     OBSWS_DEBUG_NONE = 0,       /* No debug output - production mode */
00136     OBSWS_DEBUG_LOW = 1,        /* Connection events, auth success/failure, major state changes */
00137     OBSWS_DEBUG_MEDIUM = 2,     /* Low + WebSocket opcodes, event type names, request IDs */
00138     OBSWS_DEBUG_HIGH = 3        /* Medium + full message contents - can include passwords!  */
00139 } obsws_debug_level_t;
00140
00141 /* Forward declaration of connection handle - opaque structure for connection management */
00142 typedef struct obsws_connection obsws_connection_t;
00143
00144 /**
00145 * Log callback function type - called when the library generates log messages.
00146 *
00147 * This callback gives you a chance to handle logging however you want - write to a
00148 * file, display in a GUI, send to a remote server, etc.  If you don't provide a log
00149 * callback, messages go to stderr by default.
00150 *
00151 * @param level The severity level of this log message (error, warning, info, debug)
00152 * @param message The actual log message text (null-terminated string)
00153 * @param user_data Pointer you provided in the config - use for context
00154 *
00155 * @note The message buffer is temporary and may be freed after this callback returns.
00156 *       If you need to keep the message, copy it with strdup() or similar.
00157 * @note This callback is called from an internal thread, so if you access shared
00158 *       data structures, protect them with mutexes.
00159 * @note Avoid doing expensive operations in this callback - logging should be fast.
00160 */
00161 typedef void (*obsws_log_callback_t)(obsws_log_level_t level, const char *message, void *user_data);
00162
00163 /**
00164 * Event callback function type - called when OBS sends an event.
00165 *
00166 * OBS sends events to tell you about things happening - scene changed, recording
00167 * started, input muted, etc.  Events come as JSON in event_data.  You have to parse
00168 * it yourself using something like cJSON. We don't parse it for you because different
00169 * applications care about different events, so we save CPU by leaving parsing to you.
```

```
00170 *
00171 * @param conn The connection this event came from (useful if you have multiple OBS instances)
00172 * @param event_type String name of the event like "SceneChanged", "RecordingStateChanged"
00173 * @param event_data JSON string with event details - parse this yourself with cJSON or json-c
00174 * @param user_data Pointer you provided in the config
00175 *
00176 * @note The event_data buffer is temporary and freed after this callback returns.
00177 *       If you need to keep the data, copy the string or parse it immediately.
00178 * @note This callback is called from an internal thread, so synchronize access to
00179 *       shared data structures.
00180 * @note Don't block or do expensive work in this callback - events could pile up.
00181 *
00182 * @example Parsing an event might look like:
00183 *   cJSON *event_obj = cJSON_Parse(event_data);
00184 *   if (event_obj && cJSON_HasObjectItem(event_obj, "eventData")) {
00185 *       cJSON *data = cJSON_GetObjectItem(event_obj, "eventData");
00186 *       // Now examine event_obj to see what changed
00187 *   }
00188 *   if (event_obj) cJSON_Delete(event_obj);
00189 */
00190 typedef void (*obsws_event_callback_t)(obsws_connection_t *conn, const char *event_type, const char
      *event_data, void *user_data);
00191
00192 /**
00193 * State callback function type - called when connection state changes.
00194 *
00195 * This is how you know when the connection comes up or goes down.  Use this to
00196 * update your UI - disable buttons when disconnected, enable them when connected,
00197 * show spinners during connecting, etc.
00198 *
00199 * The callback receives both the old and new states so you can see the transition.
00200 * For example, if old_state is DISCONNECTED and new_state is CONNECTING, you might
00201 * show a "connecting..." message.  If old_state is CONNECTED and new_state is
00202 * DISCONNECTED, you might show "disconnected" and disable sending commands.
00203 *
00204 * @param conn The connection whose state changed
00205 * @param old_state Previous state (what it was before)
00206 * @param new_state Current state (what it is now)
00207 * @param user_data Pointer you provided in the config
00208 *
00209 * @note This callback is called from an internal thread, so protect shared data.
00210 * @note Don't do slow operations here - state changes should be handled quickly.
00211 *
00212 * State transitions:
00213 * - DISCONNECTED -> CONNECTING (connection attempt starting)
00214 * - CONNECTING -> AUTHENTICATING (WebSocket connected, checking auth)
00215 * - AUTHENTICATING -> CONNECTED (ready to use)
00216 * - CONNECTING -> ERROR (connection failed)
00217 * - AUTHENTICATING -> ERROR (auth failed)
00218 * - CONNECTED -> DISCONNECTED (user disconnected or connection lost)
00219 * - CONNECTED -> ERROR (unexpected connection drop)
00220 * - ERROR -> DISCONNECTED (after cleanup)
00221 * - Any state -> DISCONNECTED (when you call obsws_disconnect)
00222 */
00223 typedef void (*obsws_state_callback_t)(obsws_connection_t *conn, obsws_state_t old_state,
      obsws_state_t new_state, void *user_data);
00224
00225 /**
00226 * Connection configuration structure.
00227 *
00228 * This structure holds all the settings for connecting to OBS. You should fill this
00229 * out with your specific needs, then pass it to obsws_connect().  A good starting point
00230 * is to call obsws_config_init() which fills it with reasonable defaults, then only
00231 * change the fields you care about (usually just host, port, and password).
00232 *
00233 * Design note:  We use a config struct instead of many function parameters because
00234 * it's more flexible - adding new configuration options doesn't break existing code.
00235 * It also makes it clear what options are available.
00236 */
00237 typedef struct {
00238     /* === Connection Parameters === */
00239     const char *host;                   /* IP or hostname of OBS (e.g., "192.168.1.100" or
      "obs.example.com") */
00240     uint16_t port;                      /* OBS WebSocket server port, usually 4455, sometimes 4454
      for WSS */
00241     const char *password;               /* OBS WebSocket password from settings.  Set to NULL for no
      auth.  */
00242     bool use_ssl;                       /* Use WSS (WebSocket Secure) instead of WS. Requires OBS
      configured for SSL. */
00243
00244     /* === Timeout Settings (all in milliseconds) ===
00245 Timeouts are important to prevent hanging.  Too short and you get false failures.
00246 Too long and your app freezes.  Adjust based on network quality.  */
00247     uint32_t connect_timeout_ms;        /* How long to wait for initial TCP connection (default:
      5000) */
00248     uint32_t recv_timeout_ms;           /* How long to wait for data from OBS (default:  5000) */
00249     uint32_t send_timeout_ms;           /* How long to wait to send data to OBS (default:  5000) */
```

```
00250
00251     /* === Keep-Alive / Health Monitoring ===
00252 The library sends ping messages periodically to detect dead connections.
00253 If OBS stops responding to pings, the library will try to reconnect.  */
00254     uint32_t ping_interval_ms;          /* Send ping this often (default:  10000, 0 to disable pings)
      */
00255     uint32_t ping_timeout_ms;           /* Wait this long for pong response (default:  5000) */
00256
00257     /* === Automatic Reconnection ===
00258 If the connection dies, should we try to reconnect?  Very useful for production
00259 because networks hiccup, OBS crashes, etc.  The library uses exponential backoff
00260 to avoid hammering the server - delays double each attempt up to the max.  */
00261     bool auto_reconnect;                /* Enable automatic reconnection (default:  true) */
00262     uint32_t reconnect_delay_ms;        /* Wait this long before first reconnect (default:  1000) */
00263     uint32_t max_reconnect_delay_ms;    /* Don't wait longer than this between attempts (default:
      30000) */
00264     uint32_t max_reconnect_attempts;    /* Give up after this many attempts (0 = retry forever) */
00265
00266     /* === Callbacks ===
00267 These optional callbacks let you be notified of important events.
00268 You can leave any of them NULL if you don't care about that event type.  */
00269     obsws_log_callback_t log_callback;  /* Called when library logs something */
00270     obsws_event_callback_t event_callback;  /* Called when OBS sends an event */
00271     obsws_state_callback_t state_callback;  /* Called when connection state changes */
00272     void *user_data;                    /* Passed to all callbacks - use for context (like "this"
      pointer) */
00273 } obsws_config_t;
00274
00275 /**
00276 * Connection statistics - useful for monitoring and debugging connection health.
00277 *
00278 * These stats let you see what's happening on the connection - how many messages
00279 * have been sent/received, error counts, latency, etc.  Useful for monitoring the
00280 * connection quality, detecting if something is wrong, or just being curious about
00281 * protocol activity.  You get these by calling obsws_get_stats().
00282 */
00283 typedef struct {
00284     uint64_t messages_sent;             /* Total WebSocket messages sent to OBS (includes ping/pong)
      */
00285     uint64_t messages_received;         /* Total WebSocket messages received from OBS (includes
      events) */
00286     uint64_t bytes_sent;                /* Total bytes transmitted, useful for bandwidth monitoring
      */
00287     uint64_t bytes_received;            /* Total bytes received */
00288     uint64_t reconnect_count;           /* How many times auto-reconnect kicked in (0 if never
      disconnected) */
00289     uint64_t error_count;               /* Total errors encountered (some might be retried
      successfully) */
00290     uint64_t last_ping_ms;              /* Round-trip time of last ping - network latency indicator
      */
00291     time_t connected_since;             /* Unix timestamp of when this connection was established */
00292 } obsws_stats_t;
00293
00294 /**
00295 * Response structure for requests to OBS.
00296 *
00297 * When you send a request like obsws_set_current_scene(), you can get back a
00298 * response with the result.  The response tells you if it succeeded, and if not,
00299 * why it failed.  It might also contain response data from OBS - for example,
00300 * obsws_get_current_scene() puts the scene name in response_data as JSON.
00301 *
00302 * If you don't care about the response, you can pass NULL and not get one back.
00303 * Otherwise you must free it with obsws_response_free() when done.
00304 *
00305 * Design note:  Responses are returned as strings instead of parsed JSON to save
00306 * CPU - different callers care about different response fields, so we let them
00307 * parse what they need.  This also avoids dependency bloat.
00308 */
00309 typedef struct {
00310     bool success;                       /* true if OBS said the operation worked */
00311     int status_code;                    /* OBS status code:  100-199 = success, 600+ = error */
00312     char *error_message;                /* If success is false, this has the reason (e.g., "Scene
      does not exist") */
00313     char *response_data;                /* Raw JSON response from OBS - parse yourself with cJSON */
00314 } obsws_response_t;
00315
00316 /* ============================================================================
00317 * Library Initialization and Cleanup
00318 * ============================================================================ */
00319
00320 /**
00321 * Initialize the OBS WebSocket library.
00322 *
00323 * This must be called once, before using any other library functions.  It sets up
00324 * global state like threading primitives and initializes dependencies like
00325 * libwebsockets and OpenSSL. If you call it multiple times, subsequent calls are
00326 * ignored (thread-safe).
```

```
00327 *
00328 * Typical usage:
00329 *   if (obsws_init() != OBSWS_OK) {
00330 *       fprintf(stderr, "Failed to init library\n");
00331 *       return 1;
00332 *   }
00333 *   // ...  use the library ...
00334 *   obsws_cleanup();
00335 *
00336 * @return OBSWS_OK on success, error code if initialization failed
00337 *
00338 * @note Call this from your main thread before spawning other threads.
00339 * @note Very thread-safe - can be called multiple times, only initializes once.
00340 */
00341 obsws_error_t obsws_init(void);
00342
00343 /**
00344 * Cleanup the OBS WebSocket library.
00345 *
00346 * Call this when done using the library to release resources.  Any connections
00347 * should be disconnected before cleanup, though the library will try to clean
00348 * them up if they're not.  After calling this, don't use any library functions
00349 * until you call obsws_init() again.
00350 *
00351 * @note It's safe to call this multiple times.
00352 * @note You should obsws_disconnect() all connections before calling this.
00353 * @note Threads should exit before calling cleanup - don't cleanup while
00354 *       callbacks are running.
00355 */
00356 void obsws_cleanup(void);
00357
00358 /**
00359 * Get the library version string.
00360 *
00361 * Returns a string like "1.0.0" indicating what version of the library you're
00362 * using.  Useful for debugging - you can log this at startup so you know which
00363 * version of the library handled a particular issue.
00364 *
00365 * @return Pointer to version string (valid until obsws_cleanup is called)
00366 */
00367 const char* obsws_version(void);
00368
00369 /**
00370 * Set the global log level for the library.
00371 *
00372 * This affects all library output - messages at this level and below are shown.
00373 * For example, if you set OBSWS_LOG_WARNING, you'll see warnings and errors but
00374 * not info or debug messages.  This setting affects all connections.
00375 *
00376 * Common usage:
00377 *   - Development:  OBSWS_LOG_DEBUG - see everything while developing
00378 *   - Testing:  OBSWS_LOG_INFO - see what's happening
00379 *   - Production:  OBSWS_LOG_ERROR or OBSWS_LOG_WARNING - only see problems
00380 *
00381 * @param level Minimum log level to output
00382 */
00383 void obsws_set_log_level(obsws_log_level_t level);
00384
00385 /**
00386 * Set the global debug level for the library.
00387 *
00388 * This controls detailed internal logging separate from regular log messages.
00389 * Use this for troubleshooting connection/protocol issues.  Each level includes
00390 * all output from lower levels.
00391 *
00392 * Debug output shows protocol-level information - opcodes, message IDs, etc.
00393 * It's verbose and should only be used during development.
00394 *
00395 * WARNING: Level HIGH logs passwords and raw messages.  Never use in production
00396 * or with untrusted users watching the output.
00397 *
00398 * @param level Debug verbosity level
00399 */
00400 void obsws_set_debug_level(obsws_debug_level_t level);
00401
00402 /**
00403 * Get the current global debug level.
00404 *
00405 * Returns what the debug level is set to.  Useful if you have code that needs to
00406 * know how verbose the debugging is.
00407 *
00408 * @return Current debug level
00409 */
00410 obsws_debug_level_t obsws_get_debug_level(void);
00411
00412 /* ========================================================================
00413 * Connection Management
```

```
00414 * ========================================================================== */
00415
00416 /**
00417 * Create a default configuration structure with reasonable defaults.
00418 *
00419 * This initializes a config structure with settings that should work for most
00420 * cases.  You then modify only the fields you care about.  This is better than
00421 * manually setting each field because if we add new config options in the future,
00422 * you'll automatically get the right defaults without changing your code.
00423 *
00424 * After calling this, typically you'd set:
00425 *   config.host = "192.168.1.100";
00426 *   config.port = 4455;
00427 *   config.password = "your-obs-password";
00428 *
00429 * Then pass it to obsws_connect().
00430 *
00431 * @param config Pointer to configuration structure to fill with defaults
00432 */
00433 void obsws_config_init(obsws_config_t *config);
00434
00435 /**
00436 * Create a new OBS WebSocket connection and start connecting.
00437 *
00438 * This function creates a connection object and begins the connection process
00439 * in the background.  The connection goes through states:  DISCONNECTED ->
00440 * CONNECTING -> AUTHENTICATING -> CONNECTED. You'll be notified of state
00441 * changes via the state callback (if you provided one in config).
00442 *
00443 * This is non-blocking - it returns immediately.  The actual connection happens
00444 * in a background thread.  Check obsws_get_state() to see when it reaches CONNECTED.
00445 *
00446 * Design note:  We do connection in the background because it can take a while
00447 * (DNS lookups, TCP handshake, authentication).  Blocking would freeze your app.
00448 *
00449 * @param config Connection configuration (required, cannot be NULL)
00450 * @return Connection handle on success (never NULL if called with valid config),
00451 *         NULL on failure (usually out of memory)
00452 *
00453 * @note The config structure is copied internally, so you can free or reuse it
00454 *       after calling obsws_connect().
00455 * @note The returned handle is opaque - use it with other obsws_* functions.
00456 * @note Connection attempt happens asynchronously - wait for state callback or
00457 *       call obsws_get_state() to check when it's ready.
00458 * @note If auto_reconnect is enabled in config, the library will automatically
00459 *       try to reconnect if the connection drops.
00460 *
00461 * @example Usage:
00462 *   obsws_config_t config;
00463 *   obsws_config_init(&config);
00464 *   config.host = "192.168.1.100";
00465 *   config.port = 4455;
00466 *   config.password = "mypassword";
00467 *   config.state_callback = my_state_callback;
00468 *   config.user_data = my_app_context;
00469 *
00470 *   obsws_connection_t *conn = obsws_connect(&config);
00471 *   if (!conn) {
00472 *       fprintf(stderr, "Failed to create connection\n");
00473 *       return 1;
00474 *   }
00475 *   // Wait for state callback to indicate CONNECTED...
00476 */
00477 obsws_connection_t* obsws_connect(const obsws_config_t *config);
00478
00479 /**
00480 * Disconnect and destroy a connection.
00481 *
00482 * This closes the connection to OBS and releases all associated resources.
00483 * After calling this, the connection handle is invalid - don't use it anymore.
00484 *
00485 * If auto_reconnect was enabled, it stops trying to reconnect.  If you want to
00486 * connect again, create a new connection with obsws_connect().
00487 *
00488 * This function blocks until the background thread cleanly shuts down.  It should
00489 * be fast (under 100ms), but avoid calling it from callbacks since callbacks
00490 * run in the connection thread - this would deadlock.
00491 *
00492 * @param conn Connection handle to destroy (can be NULL, which does nothing)
00493 *
00494 * @note Safe to call even if already disconnected.
00495 * @note Don't call from inside callbacks (they run in the connection thread).
00496 * @note After calling this, all pending requests are abandoned (responses never arrive).
00497 */
00498 void obsws_disconnect(obsws_connection_t *conn);
00499
00500 /**
```

```
00501 * Check if connection is currently authenticated and ready to use.
00502 *
00503 * Returns true only if the state is CONNECTED - meaning authentication completed
00504 * and you can send commands.  Returns false in all other states (DISCONNECTED,
00505 * CONNECTING, AUTHENTICATING, ERROR).
00506 *
00507 * This is the main function to check before sending commands.  If it returns false,
00508 * commands will fail with OBSWS_ERROR_NOT_CONNECTED.
00509 *
00510 * @param conn Connection handle
00511 * @return true if fully connected and authenticated, false otherwise
00512 *
00513 * @note Fast operation - just checks internal state variable.
00514 * @note Prefer using state callbacks to be notified of changes instead of polling.
00515 */
00516 bool obsws_is_connected(const obsws_connection_t *conn);
00517
00518 /**
00519 * Get the detailed current connection state.
00520 *
00521 * Similar to obsws_is_connected() but returns the full state, not just a boolean.
00522 * This lets you distinguish between different states - for example, you might show
00523 * "connecting..." if the state is CONNECTING, vs "disconnected" if DISCONNECTED.
00524 *
00525 * @param conn Connection handle
00526 * @return Current connection state (see obsws_state_t for the state machine)
00527 */
00528 obsws_state_t obsws_get_state(const obsws_connection_t *conn);
00529
00530 /**
00531 * Get connection statistics and performance metrics.
00532 *
00533 * Returns counters showing what's happened on this connection - messages sent,
00534 * bytes transmitted, errors, reconnect attempts, etc.  Useful for monitoring,
00535 * debugging, or just curiosity.
00536 *
00537 * The latency field (last_ping_ms) shows the round-trip time of the last ping,
00538 * which indicates network quality.  Higher values mean slower network or more
00539 * congestion.
00540 *
00541 * @param conn Connection handle
00542 * @param stats Pointer to structure to fill (required)
00543 * @return OBSWS_OK on success, error code if conn is invalid
00544 *
00545 * @note fast operation - just copies the stats struct.
00546 */
00547 obsws_error_t obsws_get_stats(const obsws_connection_t *conn, obsws_stats_t *stats);
00548
00549 /**
00550 * Manually trigger a reconnection attempt.
00551 *
00552 * Normally the library handles reconnection automatically if auto_reconnect is
00553 * enabled.  This function lets you force a reconnect attempt right now, for example
00554 * if you detect the connection seems dead even though the library hasn't noticed yet.
00555 *
00556 * If the connection is currently connected, this disconnects and reconnects.
00557 * If it's not connected, this starts a connection attempt.
00558 *
00559 * @param conn Connection handle
00560 * @return OBSWS_OK if reconnection started, error code otherwise
00561 *
00562 * @note This is async - it returns immediately, reconnection happens in background.
00563 * @note If auto_reconnect is disabled, this still reconnects one time.
00564 */
00565 obsws_error_t obsws_reconnect(obsws_connection_t *conn);
00566
00567 /**
00568 * Send a ping and measure round-trip time to check connection health.
00569 *
00570 * Sends a WebSocket ping to OBS and waits for the pong response.  The round-trip
00571 * time tells you the network latency.  If the ping times out, it indicates a
00572 * problem - either the network is very slow or OBS is not responding.
00573 *
00574 * This is useful for:
00575 *   - Checking if the connection is alive
00576 *   - Measuring network latency
00577 *   - Detecting when a connection appears OK but OBS isn't responding
00578 *
00579 * @param conn Connection handle
00580 * @param timeout_ms How long to wait for pong (milliseconds).  Use 5000-10000 as typical.
00581 * @return Round-trip time in milliseconds if successful, negative error code if it failed
00582 *         (probably OBSWS_ERROR_TIMEOUT if OBS isn't responding)
00583 *
00584 * @note Returns immediately if not connected - doesn't attempt to connect.
00585 * @note The library also sends pings automatically at ping_interval_ms, so you
00586 *       usually don't need to call this manually.
00587 * @note A high latency (e.g., several seconds) might indicate network problems.
```

```
00588 */
00589 int obsws_ping(obsws_connection_t *conn, uint32_t timeout_ms);
00590
00591 /* ============================================================================
00592 * Scene Management
00593 * ============================================================================ */
00594
00595 /**
00596 * Switch to a specific scene in OBS.
00597 *
00598 * This tells OBS to make a different scene active.  When you switch scenes, all
00599 * sources in the new scene become visible (if their visibility is on), and sources
00600 * from the old scene disappear.  This is the main way to change what's being streamed
00601 * or recorded.
00602 *
00603 * Switching scenes is fast but not instant - OBS processes the request and sends
00604 * a SceneChanged event when complete.  If you have many sources with animations,
00605 * the transition might take a second or so.
00606 *
00607 * @param conn Connection handle (must be in CONNECTED state)
00608 * @param scene_name Name of the scene to activate (must exist in OBS, case-sensitive)
00609 * @param response Optional pointer to receive response.  If you pass a pointer to
00610 *        response pointer, you get back a response object that you must free with
00611 *        obsws_response_free().  Pass NULL if you don't care about the response.
00612 * @return OBSWS_OK if the request was sent successfully (doesn't mean OBS processed it yet)
00613 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00614 * @return OBSWS_ERROR_INVALID_PARAM if scene_name is NULL
00615 *
00616 * @note The scene must exist in OBS. If you try to switch to a non-existent scene,
00617 *       OBS will return an error in the response.
00618 * @note This is async - the function returns before OBS actually switches scenes.
00619 *       Watch for a SceneChanged event to know when it completed.
00620 * @note If you only care about success/failure, you can pass NULL for response.
00621 * @note The response contains OBS status codes - 100-199 is success, 600+ is error.
00622 *
00623 * @example Switching to a scene:
00624 *   obsws_response_t *response = NULL;
00625 *   obsws_error_t err = obsws_set_current_scene(conn, "Gaming Scene", &response);
00626 *   if (err == OBSWS_OK && response) {
00627 *       if (response->success) {
00628 *           printf("Scene switched successfully\n");
00629 *       } else {
00630 *           printf("OBS error:  %s\n", response->error_message);
00631 *       }
00632 *       obsws_response_free(response);
00633 *   }
00634 */
00635 obsws_error_t obsws_set_current_scene(obsws_connection_t *conn, const char *scene_name,
      obsws_response_t **response);
00636
00637 /**
00638 * Get the name of the currently active scene.
00639 *
00640 * Asks OBS which scene is currently shown.  The answer comes back as a string that
00641 * you provide a buffer for.  If the buffer is too small, the function fails with
00642 * OBSWS_ERROR_RECV_FAILED (or similar) because the response doesn't fit.
00643 *
00644 * This is useful to check what scene is active before switching, or to sync your
00645 * UI state with OBS (in case OBS was controlled by something else).
00646 *
00647 * @param conn Connection handle (must be in CONNECTED state)
00648 * @param scene_name Output buffer where the scene name will be written
00649 * @param buffer_size Size of the buffer (how many bytes can fit)
00650 * @return OBSWS_OK on success
00651 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00652 * @return OBSWS_ERROR_INVALID_PARAM if scene_name is NULL
00653 *
00654 * @note The buffer should be large enough for typical scene names.  OBS doesn't
00655 *       limit scene name length, but practically they're usually under 256 characters.
00656 *       256 or 512 bytes is usually safe.
00657 * @note The string written to scene_name is null-terminated.
00658 * @note This is synchronous - it waits for response before returning (blocking).
00659 *
00660 * @example Getting current scene:
00661 *   char current_scene[256];
00662 *   if (obsws_get_current_scene(conn, current_scene, sizeof(current_scene)) == OBSWS_OK) {
00663 *       printf("Current scene:  %s\n", current_scene);
00664 *   }
00665 */
00666 obsws_error_t obsws_get_current_scene(obsws_connection_t *conn, char *scene_name, size_t buffer_size);
00667
00668 /**
00669 * Get a list of all available scenes in the OBS session.
00670 *
00671 * Asks OBS for the list of all scenes it knows about.  Returns an array of scene
00672 * name strings.  You're responsible for freeing both the individual strings and
00673 * the array itself using obsws_free_scene_list().
```

```
00674 *
00675 * Useful for:
00676 *   - Building a scene switcher UI
00677 *   - Validating that a scene name exists before trying to switch to it
00678 *   - Showing what scenes are available
00679 *
00680 * @param conn Connection handle (must be in CONNECTED state)
00681 * @param scenes Output parameter - receives pointer to array of scene name strings.
00682 *        Each string is allocated with malloc and null-terminated.  The array itself
00683 *        is also malloc'd.  You must free with obsws_free_scene_list().
00684 * @param count Output parameter - receives the number of scenes in the array
00685 * @return OBSWS_OK on success
00686 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00687 * @return OBSWS_ERROR_INVALID_PARAM if scenes or count is NULL
00688 *
00689 * @note The returned array is heap-allocated.  Always free it with obsws_free_scene_list()
00690 *       when done, or you leak memory.  Don't use free() or free the individual
00691 *       strings manually - use the provided function.
00692 * @note This is synchronous - waits for response.
00693 * @note If OBS has no scenes (unusual), you get count=0 and scenes=pointer to empty array.
00694 *
00695 * @example Getting and using scene list:
00696 *   char **scenes = NULL;
00697 *   size_t count = 0;
00698 *   if (obsws_get_scene_list(conn, &scenes, &count) == OBSWS_OK) {
00699 *       printf("Available scenes:  %zu\n", count);
00700 *       for (size_t i = 0; i < count; i++) {
00701 *           printf("  - %s\n", scenes[i]);
00702 *       }
00703 *       obsws_free_scene_list(scenes, count);
00704 *   }
00705 */
00706 obsws_error_t obsws_get_scene_list(obsws_connection_t *conn, char ***scenes, size_t *count);
00707
00708 /**
00709 * Switch to a different scene collection.
00710 *
00711 * Scene collections are different sets of scenes.  OBS can have multiple scene
00712 * collections and you can switch between them.  When you switch collections, all
00713 * the scenes in the current collection are replaced with the scenes from the new
00714 * collection.  This is useful for different contexts (e.g., "Gaming", "IRL",
00715 * "Voiceover", etc.).
00716 *
00717 * Switching collections can take a moment because OBS needs to load all the new
00718 * scenes and their settings.  You'll get a SceneCollectionChanged event when done.
00719 *
00720 * @param conn Connection handle (must be in CONNECTED state)
00721 * @param collection_name Name of the scene collection to activate (case-sensitive,
00722 *        must exist in OBS)
00723 * @param response Optional pointer to receive response (NULL to ignore)
00724 * @return OBSWS_OK if request sent successfully
00725 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00726 * @return OBSWS_ERROR_INVALID_PARAM if collection_name is NULL
00727 *
00728 * @note Scene collection names are shown in OBS: Scene Collection dropdown.
00729 * @note Switching collections is slower than switching scenes (requires loading files).
00730 * @note This is async - function returns before OBS finishes switching.
00731 * @note If the collection doesn't exist, OBS returns an error in the response.
00732 */
00733 obsws_error_t obsws_set_scene_collection(obsws_connection_t *conn, const char *collection_name,
     obsws_response_t **response);
00734
00735 /* ============================================================================
00736 * Recording and Streaming Control
00737 * ============================================================================ */
00738
00739 /**
00740 * Start recording to disk.
00741 *
00742 * Tells OBS to begin recording the current scene and audio to the configured
00743 * output file (usually somewhere in your Videos folder).  This is separate from
00744 * streaming - you can record without streaming, stream without recording, or do both.
00745 *
00746 * The actual file path is configured in OBS settings - this library doesn't control
00747 * where the file goes or what format it uses.  Those are OBS preferences.
00748 *
00749 * @param conn Connection handle (must be in CONNECTED state)
00750 * @param response Optional pointer to receive response
00751 * @return OBSWS_OK if request sent successfully (doesn't mean recording started yet)
00752 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00753 *
00754 * @note This is async - function returns immediately, recording starts in background.
00755 * @note You'll get a RecordingStateChanged event when recording actually starts.
00756 * @note If recording is already running, OBS ignores this request (no error).
00757 * @note The recorded file format depends on OBS configuration (usually MP4 or MKV).
00758 *
00759 * @example Starting to record:
```

```
00760 *   if (obsws_start_recording(conn, NULL) == OBSWS_OK) {
00761 *       printf("Recording start request sent\n");
00762 *   }
00763 */
00764 obsws_error_t obsws_start_recording(obsws_connection_t *conn, obsws_response_t **response);
00765
00766 /**
00767 * Stop recording.
00768 *
00769 * Tells OBS to stop recording and save the file.  The saved recording will include
00770 * everything from when you called obsws_start_recording() until now.  If recording
00771 * wasn't running, OBS ignores this (no error).
00772 *
00773 * After calling this, OBS takes a moment to finalize the file (write headers, etc.),
00774 * then sends a RecordingStateChanged event when complete.  You can't immediately
00775 * open the file - wait for the event first.
00776 *
00777 * @param conn Connection handle (must be in CONNECTED state)
00778 * @param response Optional pointer to receive response
00779 * @return OBSWS_OK if request sent successfully
00780 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00781 *
00782 * @note This is async - the file finalization happens after the function returns.
00783 * @note OBS sends a RecordingStateChanged event when the file is ready.
00784 * @note Don't try to move/open the file immediately - wait for the event first.
00785 */
00786 obsws_error_t obsws_stop_recording(obsws_connection_t *conn, obsws_response_t **response);
00787
00788 /**
00789 * Start streaming to a remote server.
00790 *
00791 * Tells OBS to begin streaming the current scene and audio to the configured
00792 * streaming service (Twitch, YouTube, etc.).  The stream settings (server URL, key,
00793 * bitrate, resolution, etc.)  are all configured in OBS - this library just tells
00794 * OBS when to start and stop.
00795 *
00796 * Streaming uses the same video/audio sources as recording, but you can have
00797 * different settings (OBS can transcode differently for streaming vs recording).
00798 *
00799 * @param conn Connection handle (must be in CONNECTED state)
00800 * @param response Optional pointer to receive response
00801 * @return OBSWS_OK if request sent successfully
00802 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00803 *
00804 * @note This is async - streaming starts in background.
00805 * @note You'll get a StreamStateChanged event when streaming actually starts.
00806 * @note Streaming won't start if the streaming settings are invalid (wrong RTMP key, etc).
00807 * @note If already streaming, OBS ignores this request.
00808 * @note Network conditions determine whether streaming succeeds - bad internet = failure.
00809 *
00810 * @example Starting a stream:
00811 *   obsws_error_t err = obsws_start_streaming(conn, NULL);
00812 *   if (err == OBSWS_OK) {
00813 *       // Wait for StreamStateChanged event to confirm it started
00814 *   }
00815 */
00816 obsws_error_t obsws_start_streaming(obsws_connection_t *conn, obsws_response_t **response);
00817
00818 /**
00819 * Stop streaming.
00820 *
00821 * Tells OBS to stop streaming to the remote server.  After calling this, OBS
00822 * disconnects from the streaming server and the stream ends.  The stopped event
00823 * will tell you if the disconnect was clean or if there was an issue.
00824 *
00825 * @param conn Connection handle (must be in CONNECTED state)
00826 * @param response Optional pointer to receive response
00827 * @return OBSWS_OK if request sent successfully
00828 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00829 *
00830 * @note This is async - disconnecting from the server takes a moment.
00831 * @note OBS sends a StreamStateChanged event when streaming fully stops.
00832 * @note If not streaming, OBS ignores this request (no error).
00833 */
00834 obsws_error_t obsws_stop_streaming(obsws_connection_t *conn, obsws_response_t **response);
00835
00836 /**
00837 * Get whether OBS is currently streaming.
00838 *
00839 * Returns true if a stream is active (connected to the streaming server), false
00840 * otherwise.  The response parameter (if provided) has more detailed info like
00841 * bandwidth, frames rendered, etc.
00842 *
00843 * This is useful to check state after startup - maybe OBS was already streaming
00844 * before your app connected, and you want to know about it.
00845 *
00846 * @param conn Connection handle (must be in CONNECTED state)
```

```
00847 * @param is_streaming Output parameter - true if streaming, false if not
00848 * @param response Optional pointer to receive full response with stats
00849 * @return OBSWS_OK on success
00850 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00851 * @return OBSWS_ERROR_INVALID_PARAM if is_streaming is NULL
00852 *
00853 * @note is_streaming must be non-NULL, but response can be NULL.
00854 * @note Fast operation - just queries current state.
00855 *
00856 * @example Checking if streaming:
00857 *   bool streaming = false;
00858 *   if (obsws_get_streaming_status(conn, &streaming, NULL) == OBSWS_OK) {
00859 *       printf("Streaming:  %s\n", streaming ?  "yes" :  "no");
00860 *   }
00861 */
00862 obsws_error_t obsws_get_streaming_status(obsws_connection_t *conn, bool *is_streaming,
      obsws_response_t **response);
00863
00864 /**
00865 * Get whether OBS is currently recording.
00866 *
00867 * Returns true if recording is active, false otherwise.  Similar to
00868 * obsws_get_streaming_status() but for recording instead.
00869 *
00870 * @param conn Connection handle (must be in CONNECTED state)
00871 * @param is_recording Output parameter - true if recording, false if not
00872 * @param response Optional pointer to receive full response with stats
00873 * @return OBSWS_OK on success
00874 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00875 * @return OBSWS_ERROR_INVALID_PARAM if is_recording is NULL
00876 *
00877 * @note is_recording must be non-NULL, but response can be NULL.
00878 */
00879 obsws_error_t obsws_get_recording_status(obsws_connection_t *conn, bool *is_recording,
      obsws_response_t **response);
00880
00881 /* ============================================================================
00882 * Source Control
00883 * ============================================================================ */
00884
00885 /**
00886 * Set whether a source is visible in a scene.
00887 *
00888 * Sources are the building blocks of scenes - they can be cameras, images, text,
00889 * browser windows, etc.  Each source appears in one or more scenes, and you can
00890 * control whether it's shown or hidden.  When you hide a source, it doesn't render
00891 * on the stream/recording until you show it again.
00892 *
00893 * This is useful for:
00894 *   - Show/hide a watermark or banner
00895 *   - Toggle overlays on and off
00896 *   - Control which camera appears (if you have multiple cameras as sources)
00897 *
00898 * Note:  A source can exist in multiple scenes.  Changing visibility in one scene
00899 * doesn't affect it in other scenes.
00900 *
00901 * @param conn Connection handle (must be in CONNECTED state)
00902 * @param scene_name Name of the scene containing the source (case-sensitive)
00903 * @param source_name Name of the source to hide/show (case-sensitive)
00904 * @param visible true to show the source, false to hide it
00905 * @param response Optional pointer to receive response
00906 * @return OBSWS_OK if request sent successfully
00907 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00908 * @return OBSWS_ERROR_INVALID_PARAM if scene_name or source_name is NULL
00909 *
00910 * @note If the scene or source doesn't exist, OBS returns an error in response.
00911 * @note Changes are instant - the source appears/disappears on stream immediately.
00912 * @note You'll get a SourceVisibilityChanged event when this completes.
00913 *
00914 * @example Hiding a watermark source:
00915 *   obsws_set_source_visibility(conn, "Main Scene", "Watermark", false, NULL);
00916 */
00917 obsws_error_t obsws_set_source_visibility(obsws_connection_t *conn, const char *scene_name,
00918                                           const char *source_name, bool visible, obsws_response_t
      **response);
00919
00920 /**
00921 * Enable or disable a filter on a source.
00922 *
00923 * Filters are effects applied to sources - color correction, blur, noise suppression,
00924 * etc.  Each filter can be enabled or disabled.  Disabling a filter removes its effect
00925 * without deleting it, so you can toggle it back on later.
00926 *
00927 * Use this to:
00928 *   - Dynamically control effects (blur camera when not looking at it)
00929 *   - Toggle noise suppression on/off for a microphone source
00930 *   - Enable/disable color correction for different lighting conditions
```

```
00931 *
00932 * @param conn Connection handle (must be in CONNECTED state)
00933 * @param source_name Name of the source containing the filter (case-sensitive)
00934 * @param filter_name Name of the filter to enable/disable (case-sensitive)
00935 * @param enabled true to enable the filter, false to disable it
00936 * @param response Optional pointer to receive response
00937 * @return OBSWS_OK if request sent successfully
00938 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00939 * @return OBSWS_ERROR_INVALID_PARAM if source_name or filter_name is NULL
00940 *
00941 * @note If the source or filter doesn't exist, OBS returns an error.
00942 * @note Changes are instant on stream/recording.
00943 * @note You'll get a SourceFilterEnabledStateChanged event when complete.
00944 *
00945 * @example Disabling noise suppression:
00946 *   obsws_set_source_filter_enabled(conn, "Mic", "Noise Suppression", false, NULL);
00947 */
00948 obsws_error_t obsws_set_source_filter_enabled(obsws_connection_t *conn, const char *source_name,
00949                                               const char *filter_name, bool enabled, obsws_response_t
     **response);
00950
00951 /* ============================================================================
00952 * Custom Requests
00953 * ============================================================================ */
00954
00955 /**
00956 * Send a custom request to OBS using the WebSocket protocol.
00957 *
00958 * This is the escape hatch for accessing any OBS WebSocket v5 API that the library
00959 * doesn't have a convenience function for.  You specify the request type (like
00960 * "GetVersion", "SetSourceName", etc.)  and optionally provide request data as a
00961 * JSON string.  The response comes back as JSON that you parse yourself.
00962 *
00963 * The library handles all the protocol overhead - request IDs, timeouts, etc.
00964 * You just provide the high-level request type and parameters.
00965 *
00966 * Why provide this instead of wrapping everything?  Because the OBS API has many
00967 * functions, and we wanted to keep the library focused on the most common operations.
00968 * This gives you access to everything else without bloating the library.
00969 *
00970 * @param conn Connection handle (must be in CONNECTED state)
00971 * @param request_type OBS request type name (e.g., "GetVersion", "SetSourceName")
00972 *        See OBS WebSocket v5 documentation for complete list
00973 * @param request_data JSON string with request parameters, or NULL if no parameters needed.
00974 *        For example:  "{\"sceneName\":  \"Main\", \"sourceName\":  \"Camera\"}"
00975 * @param response Pointer to receive response object (must be freed with obsws_response_free)
00976 *        If you pass NULL, the request is still sent but you don't get a response back.
00977 * @param timeout_ms How long to wait for a response (milliseconds).  0 = use default timeout.
00978 * @return OBSWS_OK if request was sent and response received within timeout
00979 * @return OBSWS_ERROR_NOT_CONNECTED if not connected
00980 * @return OBSWS_ERROR_TIMEOUT if no response within timeout_ms
00981 * @return OBSWS_ERROR_INVALID_PARAM if request_type is NULL
00982 * @return OBSWS_ERROR_PARSE_FAILED if response JSON was malformed
00983 *
00984 * @note You must know the OBS WebSocket v5 API to use this effectively.  See
00985 *       https://github.com/obsproject/obs-websocket/blob/master/docs/generated/protocol.md
00986 * @note The response_data field in the response contains the OBS response as a
00987 *       JSON string.  You parse it with cJSON or similar.
00988 * @note If response is NULL, this becomes fire-and-forget - useful for commands
00989 *       where you don't care about the result.
00990 * @note Timeout of 0 uses the global receive timeout from config (usually 5000ms).
00991 *
00992 * @example Getting OBS version:
00993 *   obsws_response_t *response = NULL;
00994 *   obsws_error_t err = obsws_send_request(conn, "GetVersion", NULL, &response, 5000);
00995 *   if (err == OBSWS_OK && response) {
00996 *       // Parse response->response_data as JSON to get version info
00997 *       cJSON *json = cJSON_Parse(response->response_data);
00998 *       // ...  extract version fields ...
00999 *       cJSON_Delete(json);
01000 *       obsws_response_free(response);
01001 *   }
01002 *
01003 * @example Setting a source name (with parameters):
01004 *   obsws_response_t *response = NULL;
01005 *   const char *params = "{\"sourceName\":  \"OldName\", \"newName\":  \"NewName\"}";
01006 *   obsws_send_request(conn, "SetSourceName", params, &response, 5000);
01007 *   if (response) {
01008 *       printf("Success:  %d\n", response->success);
01009 *       obsws_response_free(response);
01010 *   }
01011 */
01012 obsws_error_t obsws_send_request(obsws_connection_t *conn, const char *request_type,
01013                                  const char *request_data, obsws_response_t **response, uint32_t
     timeout_ms);
01014
01015 /* ============================================================================
```

```
01016 * Event Handling
01017 * ========================================================================== */
01018
01019 /**
01020 * Process pending events from the WebSocket connection.
01021 *
01022 * The library processes events in a background thread and calls your callbacks
01023 * as events arrive.  This function is provided for API compatibility and for
01024 * applications that prefer to do event processing in the main loop rather than
01025 * in background threads.
01026 *
01027 * In most cases, you don't need to call this - just set up your callbacks and
01028 * the library handles everything.  Call this only if you want explicit control
01029 * over when event processing happens (e.g., in a game loop).
01030 *
01031 * Note:  Even if you don't call this function, events are still processed in the
01032 * background thread and callbacks are still called.  This function is optional.
01033 *
01034 * @param conn Connection handle
01035 * @param timeout_ms Maximum time to wait for events (milliseconds).  0 = return immediately
01036 *        without processing, non-zero = wait up to this long for events.
01037 * @return Number of events processed, or negative error code
01038 *
01039 * @note This function is called automatically in the background thread, so you
01040 *       don't typically need to call it manually.
01041 * @note Callbacks are called from within this function (or from the background thread).
01042 * @note If you call this frequently with timeout_ms=0, you'll busy-wait (CPU usage).
01043 * @note This is provided mainly for API flexibility - most code should not use it.
01044 */
01045 int obsws_process_events(obsws_connection_t *conn, uint32_t timeout_ms);
01046
01047 /* ========================================================================
01048 * Utility Functions
01049 * ========================================================================== */
01050
01051 /**
01052 * Free a response structure and all its allocated memory.
01053 *
01054 * When you get a response from functions like obsws_set_current_scene() or
01055 * obsws_send_request(), you're responsible for freeing it when done.  This
01056 * function frees the response_data and error_message strings, plus the response
01057 * struct itself.
01058 *
01059 * Always call this when you're done with a response, or you leak memory.  It's
01060 * safe to call with NULL (does nothing).
01061 *
01062 * @param response Response to free (can be NULL)
01063 *
01064 * @note Safe to call with NULL.
01065 * @note After calling this, don't access the response pointer anymore.
01066 * @note Don't call free() manually on responses - use this function.
01067 */
01068 void obsws_response_free(obsws_response_t *response);
01069
01070 /**
01071 * Get a human-readable string for an error code.
01072 *
01073 * Converts error codes like OBSWS_ERROR_AUTH_FAILED into strings like
01074 * "Authentication failed".  Useful for logging and error messages.  The returned
01075 * strings are static (not allocated) so don't free them.
01076 *
01077 * @param error Error code to convert
01078 * @return Pointer to error string (e.g., "Invalid parameter", "Connection failed")
01079 *         Never returns NULL - unknown error codes get "Unknown error"
01080 *
01081 * @note The returned string is static and valid for the lifetime of the program.
01082 * @note Don't free the returned pointer.
01083 *
01084 * @example Logging an error:
01085 *   obsws_error_t err = obsws_set_current_scene(conn, "Scene", NULL);
01086 *   if (err != OBSWS_OK) {
01087 *       fprintf(stderr, "Error:  %s\n", obsws_error_string(err));
01088 *   }
01089 */
01090 const char* obsws_error_string(obsws_error_t error);
01091
01092 /**
01093 * Get a human-readable string for a connection state.
01094 *
01095 * Converts connection state enums like OBSWS_STATE_CONNECTED into strings like
01096 * "Connected".  Useful for debug output and status displays.
01097 *
01098 * @param state Connection state to convert
01099 * @return Pointer to state string (e.g., "Disconnected", "Connecting", "Connected")
01100 *         Never returns NULL - unknown states get "Unknown"
01101 *
01102 * @note The returned string is static, don't free it.
```

```
01103 *
01104 * @example Displaying connection status:
01105 *   obsws_state_t state = obsws_get_state(conn);
01106 *   printf("Connection state:  %s\n", obsws_state_string(state));
01107 */
01108 const char* obsws_state_string(obsws_state_t state);
01109
01110 /**
01111 * Free a scene list array returned by obsws_get_scene_list().
01112 *
01113 * When you call obsws_get_scene_list(), it allocates memory for the scene names
01114 * and the array itself.  You must free all of this using this function when done.
01115 * Don't try to free the individual strings manually or use plain free() - that
01116 * won't work correctly.
01117 *
01118 * Why have a special function for this instead of just free()?  Because the memory
01119 * layout needs special handling - there's an array of pointers, each pointing to
01120 * separately allocated strings.
01121 *
01122 * @param scenes Array of scene name strings (from obsws_get_scene_list)
01123 * @param count Number of scenes in the array
01124 *
01125 * @note Safe to call with NULL scenes pointer (does nothing).
01126 * @note After calling this, the scenes pointer is invalid - don't use it anymore.
01127 * @note The count parameter must match what obsws_get_scene_list() returned.
01128 *
01129 * @example Using and freeing scene list:
01130 *   char **scenes = NULL;
01131 *   size_t count = 0;
01132 *   if (obsws_get_scene_list(conn, &scenes, &count) == OBSWS_OK) {
01133 *       for (size_t i = 0; i < count; i++) {
01134 *           printf("Scene:  %s\n", scenes[i]);
01135 *       }
01136 *       obsws_free_scene_list(scenes, count);  // Must do this!
01137 *   }
01138 */
01139 void obsws_free_scene_list(char **scenes, size_t count);
01140
01141 #ifdef __cplusplus
01142 }
01143 #endif
01144
01145 #endif // LIBWSV5_LIBRARY_H
```

Chapter 5

Example Documentation

5.1 Parsing

Event callback function type - called when OBS sends an event.

Event callback function type - called when OBS sends an event.OBS sends events to tell you about things happening - scene changed, recording started, input muted, etc. Events come as JSON in event_data. You have to parse it yourself using something like cJSON. We don't parse it for you because different applications care about different events, so we save CPU by leaving parsing to you.

Parameters

conn	The connection this event came from (useful if you have multiple OBS instances)
event_type	String name of the event like "SceneChanged", "RecordingStateChanged"
event_data	JSON string with event details - parse this yourself with cJSON or json-c
user_data	Pointer you provided in the config

Note

The event_data buffer is temporary and freed after this callback returns. If you need to keep the data, copy the string or parse it immediately.

This callback is called from an internal thread, so synchronize access to shared data structures.

Don't block or do expensive work in this callback - events could pile up.

an event might look like: cJSON ∗event_obj = cJSON_Parse(event_data); if (event_obj && cJSON_HasObject↩ Item(event_obj, "eventData")) { cJSON ∗data = cJSON_GetObjectItem(event_obj, "eventData"); // Now examine event_obj to see what changed } if (event_obj) cJSON_Delete(event_obj);

5.2 Usage

Create a new OBS WebSocket connection and start connecting.

Create a new OBS WebSocket connection and start connecting.This function creates a connection object and begins the connection process in the background. The connection goes through states: DISCONNECTED ->

CONNECTING -> AUTHENTICATING -> CONNECTED. You'll be notified of state changes via the state callback (if you provided one in config).

This is non-blocking - it returns immediately. The actual connection happens in a background thread. Check obsws_get_state() to see when it reaches CONNECTED.

Design note: We do connection in the background because it can take a while (DNS lookups, TCP handshake, authentication). Blocking would freeze your app.

Parameters

config	Connection configuration (required, cannot be NULL)

Returns

Connection handle on success (never NULL if called with valid config), NULL on failure (usually out of memory)

Note

The config structure is copied internally, so you can free or reuse it after calling obsws_connect().

The returned handle is opaque - use it with other obsws_∗ functions.

Connection attempt happens asynchronously - wait for state callback or call obsws_get_state() to check when it's ready.

If auto_reconnect is enabled in config, the library will automatically try to reconnect if the connection drops.

: obsws_config_t config; obsws_config_init(&config); config.host = "192.168.1.100"; config.port = 4455; config.↩ password = "mypassword"; config.state_callback = my_state_callback; config.user_data = my_app_context;

obsws_connection_t ∗conn = obsws_connect(&config); if (!conn) { fprintf(stderr, "Failed to create connection\n"); return 1; } // Wait for state callback to indicate CONNECTED...

5.3 Switching

Switch to a specific scene in OBS.

Switch to a specific scene in OBS.This tells OBS to make a different scene active. When you switch scenes, all sources in the new scene become visible (if their visibility is on), and sources from the old scene disappear. This is the main way to change what's being streamed or recorded.

Switching scenes is fast but not instant - OBS processes the request and sends a SceneChanged event when complete. If you have many sources with animations, the transition might take a second or so.

Parameters

conn	Connection handle (must be in CONNECTED state)
scene_name	Name of the scene to activate (must exist in OBS, case-sensitive)
response	Optional pointer to receive response. If you pass a pointer to response pointer, you get back a response object that you must free with obsws_response_free(). Pass NULL if you don't care about the response.

Returns

OBSWS_OK if the request was sent successfully (doesn't mean OBS processed it yet)

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if scene_name is NULL

Note

The scene must exist in OBS. If you try to switch to a non-existent scene, OBS will return an error in the response.

This is async - the function returns before OBS actually switches scenes. Watch for a SceneChanged event to know when it completed.

If you only care about success/failure, you can pass NULL for response.

The response contains OBS status codes - 100-199 is success, 600+ is error.

to a scene: obsws_response_t ∗response = NULL; obsws_error_t err = obsws_set_current_scene(conn, "Gaming Scene", &response); if (err == OBSWS_OK && response) { if (response->success) { printf("Scene switched successfully\n"); } else { printf("OBS error: %s\n", response->error_message); } obsws_response_free(response); }

5.4 Getting

Get the name of the currently active scene.

Get the name of the currently active scene.Asks OBS which scene is currently shown. The answer comes back as a string that you provide a buffer for. If the buffer is too small, the function fails with OBSWS_ERROR_RECV_FAILED (or similar) because the response doesn't fit.

This is useful to check what scene is active before switching, or to sync your UI state with OBS (in case OBS was controlled by something else).

Parameters

conn	Connection handle (must be in CONNECTED state)
scene_name	Output buffer where the scene name will be written
buffer_size	Size of the buffer (how many bytes can fit)

Returns

OBSWS_OK on success

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if scene_name is NULL

Note

The buffer should be large enough for typical scene names. OBS doesn't limit scene name length, but practically they're usually under 256 characters. 256 or 512 bytes is usually safe.

The string written to scene_name is null-terminated.

This is synchronous - it waits for response before returning (blocking).

current scene: char current_scene[256]; if (obsws_get_current_scene(conn, current_scene, sizeof(current_scene)) == OBSWS_OK) { printf("Current scene: %s\n", current_scene); }

5.5 Starting

Start recording to disk.

Start recording to disk.Tells OBS to begin recording the current scene and audio to the configured output file (usually somewhere in your Videos folder). This is separate from streaming - you can record without streaming, stream without recording, or do both.

The actual file path is configured in OBS settings - this library doesn't control where the file goes or what format it uses. Those are OBS preferences.

Parameters

conn	Connection handle (must be in CONNECTED state)
response	Optional pointer to receive response

Returns

OBSWS_OK if request sent successfully (doesn't mean recording started yet)

OBSWS_ERROR_NOT_CONNECTED if not connected

Note

This is async - function returns immediately, recording starts in background.

You'll get a RecordingStateChanged event when recording actually starts.

If recording is already running, OBS ignores this request (no error).

The recorded file format depends on OBS configuration (usually MP4 or MKV).

to record: if (obsws_start_recording(conn, NULL) == OBSWS_OK) { printf("Recording start request sent\n"); }

5.6 Checking

Get whether OBS is currently streaming.

Get whether OBS is currently streaming.Returns true if a stream is active (connected to the streaming server), false otherwise. The response parameter (if provided) has more detailed info like bandwidth, frames rendered, etc.

This is useful to check state after startup - maybe OBS was already streaming before your app connected, and you want to know about it.

Parameters

conn	Connection handle (must be in CONNECTED state)
is_streaming	Output parameter - true if streaming, false if not
response	Optional pointer to receive full response with stats

Returns

OBSWS_OK on success

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if is_streaming is NULL

Note

is_streaming must be non-NULL, but response can be NULL.

Fast operation - just queries current state.

if streaming: bool streaming = false; if (obsws_get_streaming_status(conn, &streaming, NULL) == OBSWS_OK) { printf("Streaming: %s\n", streaming ? "yes" : "no"); }

5.7 Hiding

Set whether a source is visible in a scene.

Set whether a source is visible in a scene.Sources are the building blocks of scenes - they can be cameras, images, text, browser windows, etc. Each source appears in one or more scenes, and you can control whether it's shown or hidden. When you hide a source, it doesn't render on the stream/recording until you show it again.

This is useful for:

- Show/hide a watermark or banner
- Toggle overlays on and off
- Control which camera appears (if you have multiple cameras as sources)

Note: A source can exist in multiple scenes. Changing visibility in one scene doesn't affect it in other scenes.

Parameters

conn	Connection handle (must be in CONNECTED state)
scene_name	Name of the scene containing the source (case-sensitive)
source_name	Name of the source to hide/show (case-sensitive)
visible	true to show the source, false to hide it
response	Optional pointer to receive response

Returns

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if scene_name or source_name is NULL

Note

If the scene or source doesn't exist, OBS returns an error in response.

Changes are instant - the source appears/disappears on stream immediately.

You'll get a SourceVisibilityChanged event when this completes.

a watermark source: obsws_set_source_visibility(conn, "Main Scene", "Watermark", false, NULL);

5.8 Disabling

Enable or disable a filter on a source.

Enable or disable a filter on a source.Filters are effects applied to sources - color correction, blur, noise suppression, etc. Each filter can be enabled or disabled. Disabling a filter removes its effect without deleting it, so you can toggle it back on later.

Use this to:

- Dynamically control effects (blur camera when not looking at it)
- Toggle noise suppression on/off for a microphone source
- Enable/disable color correction for different lighting conditions

Parameters

conn	Connection handle (must be in CONNECTED state)
source_name	Name of the source containing the filter (case-sensitive)
filter_name	Name of the filter to enable/disable (case-sensitive)
enabled	true to enable the filter, false to disable it
response	Optional pointer to receive response

Returns

OBSWS_OK if request sent successfully

OBSWS_ERROR_NOT_CONNECTED if not connected

OBSWS_ERROR_INVALID_PARAM if source_name or filter_name is NULL

Note

If the source or filter doesn't exist, OBS returns an error.

Changes are instant on stream/recording.

You'll get a SourceFilterEnabledStateChanged event when complete.

noise suppression: obsws_set_source_filter_enabled(conn, "Mic", "Noise Suppression", false, NULL);

5.9 Setting

a source name (with parameters): obsws_response_t *response = NULL; const char *params = "{\"sourceName\": \"OldName\", \"newName\": \"NewName\"}"; obsws_send_request(conn, "SetSourceName", params, &response, 5000); if (response) { printf("Success: %d\n", response->success); obsws_response_free(response); }

a source name (with parameters): obsws_response_t *response = NULL; const char *params = "{\"sourceName\": \"OldName\", \"newName\": \"NewName\"}"; obsws_send_request(conn, "SetSourceName", params, &response, 5000); if (response) { printf("Success: %d\n", response->success); obsws_response_free(response); }

5.10 Logging

Get a human-readable string for an error code.

Get a human-readable string for an error code.Converts error codes like OBSWS_ERROR_AUTH_FAILED into strings like "Authentication failed". Useful for logging and error messages. The returned strings are static (not allocated) so don't free them.

Parameters

error	Error code to convert

Returns

Pointer to error string (e.g., "Invalid parameter", "Connection failed") Never returns NULL - unknown error codes get "Unknown error"

Note

The returned string is static and valid for the lifetime of the program.

Don't free the returned pointer.

an error: obsws_error_t err = obsws_set_current_scene(conn, "Scene", NULL); if (err != OBSWS_OK) { fprintf(stderr, "Error: %s\n", obsws_error_string(err)); }

5.11 Displaying

Get a human-readable string for a connection state.

Get a human-readable string for a connection state.Converts connection state enums like OBSWS_STATE_↩ CONNECTED into strings like "Connected". Useful for debug output and status displays.

Parameters

state	Connection state to convert

Returns

Pointer to state string (e.g., "Disconnected", "Connecting", "Connected") Never returns NULL - unknown states get "Unknown"

Note

The returned string is static, don't free it.

connection status: obsws_state_t state = obsws_get_state(conn); printf("Connection state: %s\n", obsws_state_↩ string(state));

5.12 Using

Free a scene list array returned by obsws_get_scene_list().

Free a scene list array returned by obsws_get_scene_list().When you call obsws_get_scene_list(), it allocates memory for the scene names and the array itself. You must free all of this using this function when done. Don't try to free the individual strings manually or use plain free() - that won't work correctly.

Why have a special function for this instead of just free()? Because the memory layout needs special handling - there's an array of pointers, each pointing to separately allocated strings.

Parameters

scenes	Array of scene name strings (from obsws_get_scene_list)
count	Number of scenes in the array

Note

Safe to call with NULL scenes pointer (does nothing).

After calling this, the scenes pointer is invalid - don't use it anymore.

The count parameter must match what obsws_get_scene_list() returned.

and freeing scene list: char ∗∗scenes = NULL; size_t count = 0; if (obsws_get_scene_list(conn, &scenes, &count) == OBSWS_OK) { for (size_t i = 0; i < count; i++) { printf("Scene: %s\n", scenes[i]); } obsws_free_scene_list(scenes, count); // Must do this! }