SRT API Functions

- **Library Initialization**
    - srt_startup
    - srt_cleanup
- **Creating and configuring sockets**
    - srt_socket
    - srt_bind
    - srt_create_socket
    - srt_bind_peerof
    - srt_getsockstate
    - rt_getsndbuffer
    - srt_close
- **Connecting**
    - srt_listen
    - srt_accept
    - srt_listen_callback
    - srt_connect
    - srt_connect_debug
    - srt_rendezvous
- **Options and properties**
    - srt_getpeername
    - srt_getsockname
    - srt_getsockopt, srt_getsockflag
    - srt_setsockopt, srt_setsockflag
- **Helper data types for transmission**
    - SRT_MSGCTRL
- **Transmission**
    - srt_send, srt_sendmsg, srt_sendmsg2
    - srt_recv, srt_recvmsg, srt_recvmsg2
    - srt_sendfile, srt_recvfile
- **Diagnostics**
    - srt_getlasterror_str
    - srt_getlasterror
    - srt_strerror
    - srt_clearlasterror
- **Performance tracking**
    - srt_bstats, srt_bistats
- **Asynchronous operations (epoll)**
    - srt_epoll_create
    - srt_epoll_add_usock, srt_epoll_add_ssock, srt_epoll_update_usock, srt_epoll_update_ssock
    - srt_epoll_remove_usock, srt_epoll_remove_ssock
    - srt_epoll_wait
    - srt_epoll_release

- **Logging control**
  - srt_setloglevel
  - srt_addlogfa, srt_dellogfa, srt_resetlogfa
  - srt_setloghandler
  - srt_setlogflags

Library initialization

srt_startup

```
int srt_startup(void);
```

This function shall be called at the start of an application that uses the SRT library. It provides all necessary platform-specific initializations, sets up global data, and starts the SRT GC thread. If this function isn't explicitly called, it will be called automatically when creating the first socket. However, relying on this behavior is strongly discouraged.

- Returns:
  - 0 = successfully run, or already started
  - 1 = this is the first startup, but the GC thread is already running
  - -1 = failed
- Errors:
  - `SRT_ECONNSETUP` (with error code set): Reported when required system resource(s) failed to initialize. This is currently used only on Windows to report a failure from `WSAStartup`.

srt_cleanup

```
int srt_cleanup(void);
```

This function cleans up all global SRT resources and shall be called just before exiting the application that uses the SRT library. This cleanup function will still be called from the C++ global destructor, if not called by the application, although relying on this behavior is stronly discouraged.

- Returns:
  - 0 (A possibility to return other values is reserved for future use)

IMPORTANT: Note that the startup/cleanup calls have an instance counter. This means that if you call `srt_startup` multiple times, you need to call the `srt_cleanup` function exactly the same number of times.

Creating and configuring sockets

srt_socket

```
SRTSOCKET srt_socket(int af, int type, int protocol);
```

Creates an SRT socket:

- `af`: family (either `AF_INET` or `AF_INET6`)
- `type`, `protocol`: ignored

NOTE: The UDT library uses the `type` parameter to specify **file** or **message mode** by specifying that `SOCK_STREAM` corresponds to a TCP-like file transmission mode, and `SOCK_DGRAM` corresponds to an SCTP-like message transmission mode. SRT still supports these modes. However, this is controlled by the `SRTO_MESSAGEAPI` socket option when the transmission type is file (`SRTO_TRANSTYPE` set to `SRTT_FILE`) and the only reasonable value for the `type` parameter here is `SOCK_DGRAM`.

- Returns:
  - a valid socket ID on success
  - `INVALID_SOCKET` (-1) on error
- Errors:
  - `SRT_ENOTBUF`: not enough memory to allocate required resources

NOTE: This is probably a design flaw (**BUG?**). Usually underlying system errors are reported by `SRT_ECONNSETUP`.

srt_create_socket

```
SRTSOCKET srt_create_socket();
```

Creates a socket in `AF_INET` family only.

NOTE: In future the address family may be removed from initial socket configuration, which will free the user from specifying it in `srt_socket`. The `srt_create_socket` function will be used for all families. The family will be specified only with the first `srt_bind` or `srt_connect`, and in this case `SRT_EINVPARAM` will not be used (see `srt_bind` below).

srt_bind

```
int srt_bind(SRTSOCKET u, const struct sockaddr* name, int namelen);
```

Binds a socket to a local address and port. Binding specifies the local network interface and the UDP port number to be used for the socket. When the local address is a form of `INADDR_ANY`, then it's bound to all interfaces. When the port number is 0, then the port number will be system-allocated if necessary.

This call is obligatory for a listening socket before calling `srt_listen` and for rendezvous mode before calling `srt_connect`, otherwise it's optional. For a listening socket it defines the network interface and the port where the listener should expect a call request. In case of rendezvous mode (when the socket has set `SRTO_RENDEZVOUS` to true, in this mode both parties connect to one another) it defines the network interface and port from which packets will be sent to the peer and to which the peer is expected to send packets.

For a connecting socket this call can set up the outgoing port to be used in the communication. It is allowed that multiple SRT sockets share one local outgoing port, as long as `SRTO_REUSEADDR` is set to *true* (default). Without this call the port will be automatically selected by the system.

*See **NOTE** below under* `srt_create_socket`.

- Returns:
  - `SRT_ERROR` (-1) on error, otherwise 0
- Errors:
  - `SRT_EINVSOCK`: Socket passed as `u` designates no valid socket
  - `SRT_EINVOP`: Socket already bound
  - `SRT_EINVPARAM`: Address family in `name` is not one set for `srt_socket`
  - `SRT_ECONNSETUP`: Internal creation of a UDP socket failed
  - `SRT_ESOCKFAIL`: Internal configuration of a UDP socket (`bind`, `setsockopt`) failed

NOTE: `SRT_EINVPARAM` will not be the case in future, when the family is removed from the initial socket configuration, see `srt_create_socket` above.

srt_bind_peerof

```
int srt_bind_peerof(SRTSOCKET u, UDPSOCKET udpsock);
```

A version of `srt_bind` that acquires a given UDP socket instead of creating one.

srt_getsockstate

```
SRT_SOCKSTATUS srt_getsockstate(SRTSOCKET u);
```

Gets the current status of the socket. Possible states are:

- `SRTS_INIT`: Created, but not bound
- `SRTS_OPENED`: Created and bound, but not in use yet.
- `SRTS_LISTENING`: Socket is in listening state
- `SRTS_CONNECTING`: The connect operation was initiated, but not yet finished. This may also mean that it has timed out; you can only know that after getting a socket error report from `srt_epoll_wait`. In blocking mode it's not possible because `srt_connect` does not return until the socket is connected or failed due to timeout or interrupted call.
- `SRTS_CONNECTED`: The socket is connected and ready for transmission.
- `SRTS_BROKEN`: The socket was connected, but the connection was broken

- `SRTS_CLOSING`: The socket may still be open and active, but closing is requested, so no further operations will be accepted (active operations will be completed before closing)
- `SRTS_CLOSED`: The socket has been closed, but not yet removed by the GC thread
- `SRTS_NONEXIST`: The specified number does not correspond to a valid socket.

srt_getsndbuffer

```
int srt_getsndbuffer(SRTSOCKET sock, size_t* blocks, size_t* bytes);
```

Retrieves information about the sender buffer.

- `sock`: Socket to test
- `blocks`: Written information about buffer blocks in use
- `bytes`: Written information about bytes in use

This function can be used for diagnostics. It is especially useful when the socket needs to be closed asynchronously.

srt_close

```
int srt_close(SRTSOCKET u);
```

Closes the socket and frees all used resources. Note that underlying UDP sockets may be shared between sockets, so these are freed only with the last user closed.

- Returns:
  - `SRT_ERROR` (-1) in case of error, otherwise 0
- Errors:
  - `SRT_EINVSOCK`: Socket `u` indicates no valid socket ID

Connecting

srt_listen

```
int srt_listen(SRTSOCKET u, int backlog);
```

This sets up the listening state on a socket with a backlog setting that defines how many sockets may be allowed to wait until they are accepted (excessive connection requests are rejected in advance).

- Returns:
  - `SRT_ERROR` (-1) in case of error, otherwise 0.

- Errors:

  - `SRT_EINVPARAM`: Value of `backlog` is 0 or negative.
  - `SRT_EINVSOCK`: Socket `u` indicates no valid SRT socket.
  - `SRT_EUNBOUNDSOCK`: `srt_bind` has not yet been called on that socket.
  - `SRT_ERDVNOSERV`: `SRTO_RENDEZVOUS` flag is set to true on specified socket.
  - `SRT_EINVOP`: Internal error (should not happen when `SRT_EUNBOUNDSOCK` is reported).
  - `SRT_ECONNSOCK`: The socket is already connected.
  - `SRT_EDUPLISTEN`: The address used in `srt_bind` by this socket is already occupied by another listening socket. Binding multiple sockets to one IP address and port is allowed, as long as `SRTO_REUSEADDR` is set to true, but only one of these sockets can be set up as a listener.

srt_accept

```
SRTSOCKET srt_accept(SRTSOCKET lsn, struct sockaddr* addr, int* addrlen);
```

Accepts a pending connection and creates a new socket to handle it. The socket that is connected to a remote party is returned.

- `lsn`: the listener socket previously configured by `srt_listen`
- `addr`: the IP address and port specification for the remote party
- `addrlen`: INPUT: size of `addr` pointed object. OUTPUT: real size of the returned object

NOTE: `addr` is allowed to be NULL, in which case it's understood that the application is not interested in the address from which the connection originated. Otherwise `addr` should specify an object into which the address will be written, and `addrlen` must also specify a variable to contain the object size.

- Returns:

  - A valid socket ID for the connection, on success
  - `SRT_ERROR` (-1) on failure

- Errors:

  - `SRT_EINVPARAM`: NULL specified as `addrlen`, when `addr` is not NULL
  - `SRT_EINVSOCK`: `lsn` designates no valid socket ID. Can also mean Internal Error when an error occurred while creating an accepted socket (**BUG?**)
  - `SRT_ENOLISTEN`: `lsn` is not set up as a listener (`srt_listen` not called, or the listener socket has already been closed)
  - `SRT_ERDVNOSERV`: Internal error (if no `SRT_ENOLISTEN` reported, it means that the socket could not be set up as rendezvous because `srt_listen` does not allow it)
  - `SRT_EASYNCRCV`: No connection reported so far. This error is reported only when the `lsn` listener socket was configured as non-blocking for reading (`SRTO_RCVSYN` set to false); otherwise the call blocks until a connection is reported or an error occurs

srt_listen_callback

```
int srt_listen_callback(SRTSOCKET lsn, srt_listen_callback_fn* hook_fn, void*
hook_opaque);
```

This call installs a callback hook, which will be executed on a socket that is automatically created to handle the incoming connection on the listeneing socket (and is about to be returned by `srt_accept`), but before the connection has been accepted.

- `lsn`: Listening socket where you want to install the callback hook
- `hook_fn`: The callback hook function pointer
- `hook_opaque`: The pointer value that will be passed to the callback function

- Returns:

  - 0, if successful
  - -1, on error

- Errors:

  - `SRT_EINVPARAM` reported when `hook_fn` is a null pointer

The callback function has the signature as per this type definition:

```
typedef int srt_listen_callback_fn(void* opaque, SRTSOCKET ns, int hs_version
             const struct sockaddr* peeraddr, const char* streamid);
```

The callback function gets the following parameters passed:

- `opaque`: The pointer passed as `hook_opaque` when registering
- `ns`: The freshly created socket to handle the incoming connection
- `hs_version`: The handshake version (usually 5, pre-1.3 versions use 4)
- `peeraddr`: The address of the incoming connection
- `streamid`: The value set to `SRTO_STREAMID` option set on the peer side

(Note that versions that use handshake version 4 are incapable of using any extensions, such as streamid, however they do support encryption. Note also that the SRT version isn't yet extracted, however you can prevent too old version connections using `SRTO_MINVERSION` option).

The callback function is given an opportunity to:

- use the passed information (streamid and peer address) to decide what to do with this connection
- alter any options on the socket, which could not be set properly before on the listening socket to be derived by the accepted socket, and won't be allowed to be altered after the socket is returned by `srt_accept`

Note that the returned socket has already set all derived options from the listener socket, as it happens normally, and the moment when this callback is called is when the conclusion handshake has been already received from the caller party, but not yet interpreted (the streamid field is extracted from it prematurely).

When you, for example, set a passphrase on the socket at this very moment, the Key Material processing will happen against this already set passphrase, after the callback function is finished.

The callback function shall return 0, if the connection is to be accepted. If you return -1, **or** if the function throws an exception, this will be understood as a request to reject the incoming connection - in which case the about-to-be-accepted socket will be silently deleted and `srt_accept` will not report it. Note that in case of non-blocking mode the epoll bits for read-ready on the listener socket will not be set if the connection is rejected, including when rejected from this user function.

IMPORTANT: This function is called in the receiver worker thread, which means that it must do its checks and operations as quickly as possible and keep the minimum possible time, as every delay you do in this function will burden the processing of the incoming data on the associated UDP socket, which in case of a listener socket means the listener socket itself and every socket accepted off this listener socket. Avoid any extensive search operations, best cache in memory whatever database you have to check against the data received in streamid or peeraddr.

srt_connect

```
int srt_connect(SRTSOCKET u, const struct sockaddr* name, int namelen);
```

Connects a socket to a remote party with a specified address and port.

- `u`: SRT socket. This must be a freshly created socket that has not yet been used for anything except possibly `srt_bind`.
- `name`: specification of the remote address and port
- `namelen`: size of the object passed by `name`

NOTES:

1. See **NOTE** regarding family under `srt_create_socket`, and `SRT_EINVPARAM` error under `srt_bind` above.
2. The socket used here may be bound from upside so that it uses a predefined network interface or local outgoing port. If not, it behaves as if it was bound to `INADDR_ANY` (which binds on all interfaces) and port 0 (which makes the system assign the port automatically).

- Returns:
  - `SRT_ERROR` (-1) in case of error, otherwise 0
- Errors:
  - `SRT_EINVSOCK`: Socket `u` indicates no valid socket ID
  - `SRT_EINVPARAM`: Address family in `name` is not one set for `srt_socket`
  - `SRT_ERDVUNBOUND`: Socket `u` has set `SRTO_RENDEZVOUS` to true, but `srt_bind` hasn't yet been called on it. The `srt_connect` function is also used to connect a rendezvous socket, but rendezvous sockets must be explicitly bound to a local interface prior to connecting. Non-rendezvous sockets (caller sockets) can be left without binding - the call to `srt_connect` will bind them automatically.

    - `SRT_ECONNSOCK`: Socket `u` is already connected
    - `SRT_ECONNREJ`: Connection has been rejected

In case when `SRT_ECONNREJ` error was reported, you can get the reason for a rejected connection from `srt_getrejectreason`.

srt_connect_debug

```
int srt_connect_debug(SRTSOCKET u, const struct sockaddr* name, int namelen, int
forced_isn);
```

This function is for developers only and can be used for testing. It does the same thing as `srt_connect`, with the exception that it allows specifying the Initial Sequence Number for data transmission. Normally this value is generated randomly.

srt_rendezvous

```
int srt_rendezvous(SRTSOCKET u, const struct sockaddr* local_name, int
local_namelen,
        const struct sockaddr* remote_name, int remote_namelen);
```

Performs a rendezvous connection. This is a shortcut for doing bind locally, setting the `SRTO_RENDEZVOUS` option to true, and doing `srt_connect`.

- `u`: socket to connect
- `local_name`: specifies the local network interface and port to bind
- `remote_name`: specifies the remote party's IP address and port

NOTE: The port value shall be the same in `local_name` and `remote_name`.

Options and properties

srt_getpeername

```
int srt_getpeername(SRTSOCKET u, struct sockaddr* name, int* namelen);
```

Retrieves the remote address to which the socket is connected.

- Returns:
    - `SRT_ERROR` (-1) in case of error, otherwise 0
- Errors:
    - `SRT_EINVSOCK`: Socket `u` indicates no valid socket ID
    - `SRT_ENOCONN`: Socket `u` isn't connected, so there's no remote address to return

srt_getsockname

```
int srt_getsockname(SRTSOCKET u, struct sockaddr* name, int* namelen);
```

Extracts the address to which the socket was bound. Although you should know the address(es) that you have used for binding yourself, this function can be useful for extracting the local outgoing port number when it was specified as 0 with binding for system autoselection. With this function you can extract the port number after it has been autoselected.

- Returns:
  - `SRT_ERROR` (-1) in case of error, otherwise 0
- Errors:
  - `SRT_EINVSOCK`: Socket `u` indicates no valid socket ID
  - `SRT_ENOCONN`: Socket `u` isn't bound, so there's no local address to return (**BUG?** It should rather be `SRT_EUNBOUNDSOCK`)

srt_getsockopt, srt_getsockflag

```
int srt_getsockopt(SRTSOCKET u, int level /*ignored*/, SRT_SOCKOPT opt, void*
optval, int* optlen);
int srt_getsockflag(SRTSOCKET u, SRT_SOCKOPT opt, void* optval, int* optlen);
```

Gets the value of the given socket option. The first version (`srt_getsockopt`) respects the BSD socket API convention, although the "level" parameter is ignored. The second version (`srt_getsockflag`) omits the "level" parameter completely.

Options correspond to various data types, so you need to know what data type is assigned to a particular option, and to pass a variable of the appropriate data type. Specifications are provided in the `apps/socketoptions.hpp` file at the `srt_options` object declaration.

- Returns:
  - `SRT_ERROR` (-1) in case of error, otherwise 0
- Errors:
  - `SRT_EINVSOCK`: Socket `u` indicates no valid socket ID
  - `SRT_EINVOP`: Option `opt` indicates no valid option

srt_setsockopt, srt_setsockflag

```
int srt_setsockopt(SRTSOCKET u, int level /*ignored*/, SRT_SOCKOPT opt, const
void* optval, int optlen);
int srt_setsockflag(SRTSOCKET u, SRT_SOCKOPT opt, const void* optval, int optlen);
```

Sets a value for a socket option. The first version (`srt_setsockopt`) respects the BSD socket API convention, although the "level" parameter is ignored. The second version (`srt_setsockflag`) omits the "level" parameter completely.

Options correspond to various data types, so you need to know what data type is assigned to a particular option, and to pass a variable of the appropriate data type with the option value to be set.

- Returns:
    - `SRT_ERROR` (-1) in case of error, otherwise 0

-Errors:

- `SRT_EINVSOCK`: Socket `u` indicates no valid socket ID
- `SRT_EINVOP`: Option `opt` indicates no valid option
- Various other errors that may result from problems when setting a specific option (see option description for details).

Helper data types for transmission

SRT_MSGCTRL

The `SRT_MSGCTRL` structure:

```
typedef struct SRT_MsgCtrl_
{
   int flags;            // Left for future
   int msgttl;           // TTL for a message, default -1 (delivered always)
   int inorder;          // Whether a message is allowed to supersede partially
lost one. Unused in stream and live mode.
   int boundary;         //0:mid pkt, 1(01b):end of frame, 2(11b):complete frame,
3(10b): start of frame
   uint64_t srctime;     // source timestamp (usec), 0: use internal time
   int32_t pktseq;       // sequence number of the first packet in received
message (unused for sending)
   int32_t msgno;        // message number (output value for both sending and
receiving)
} SRT_MSGCTRL;
```

The `SRT_MSGCTRL` structure is used in `srt_sendmsg2` and `srt_recvmsg2` calls and specifies some special extra parameters:

- `flags`: [IN, OUT]. RESERVED FOR FUTURE USE (should be 0). This is intended to specify some special options controlling the details of how the called function should work.

- `msgttl`: [IN]. In **message** and **live mode** only, specifies the TTL for sending messages (in `[ms]`). Not used for receiving messages. A packet is scheduled for sending by this call and then waits in the sender buffer to be picked up at the moment when all previously scheduled data are already sent, which may be blocked when the data are scheduled faster than the network can afford to send. Default -1 means to wait indefinitely. If specified, then the packet waits for an opportunity to be sent over the network only up to this TTL, and then, if still not sent, the packet is discarded.

- `inorder`: [IN]. In **message mode** only, specifies that sent messages should be extracted by the receiver in the order of sending. This can be meaningful if a packet loss has happened, and a particular message must wait for retransmission so that it can be reassembled and then delivered. When this flag is false, the message can be delivered even if there are any previous messages still waiting for completion.
- `boundary`: RESERVED FOR FUTURE USE. Intended to be used in a special mode when you are allowed to send or retrieve a part of the message.
- `srctime`:
  - [IN] Sender only. Specifies the application-provided timestamp. If not used (specified as 0), the current system time (absolute microseconds since epoch) is used.
  - [OUT] Receiver only. Specifies the time when the packet was intended to be delivered to the receiver.
- `pktseq`: Receiver only. Reports the sequence number for the packet carrying out the payload being returned. If the payload is carried out by more than one UDP packet, only the sequence of the first one is reported. Note that in **live mode** there's always one UDP packet per message.
- `msgno`: Message number that can be sent by both sender and receiver, although it is required that this value remain monotonic in subsequent send calls. Normally message numbers start with 1 and increase with every message sent.

Helpers for `SRT_MSGCTRL`:

```
void srt_msgctrl_init(SRT_MSGCTRL* mctrl);
const SRT_MSGCTRL srt_msgctrl_default;
```

Helpers for getting an object of `SRT_MSGCTRL` type ready to use. The first is a function that fills the object with default values. The second is a constant object and can be used as a source for assignment. Note that you cannot pass this constant object into any of the API functions because they require it to be mutable, as they use some fields to output values.

Transmission

srt_send, srt_sendmsg, srt_sendmsg2

```
int srt_send(SRTSOCKET u, const char* buf, int len);
int srt_sendmsg(SRTSOCKET u, const char* buf, int len, int ttl/* = -1*/, int
inorder/* = false*/);
int srt_sendmsg2(SRTSOCKET u, const char* buf, int len, SRT_MSGCTRL *mctrl);
```

Sends a payload to a remote party over a given socket.

- `u`: Socket used to send. The socket must be connected for this operation.
- `buf`: Points to the buffer containing the payload to send.
- `len`: Size of the payload specified in `buf`.
- `ttl`: Time (in `[ms]`) to wait for a possibility to send. See description of the `SRT_MSGCTRL::msgttl` field.
- `inorder`: Required to be received in the order of sending. See `SRT_MSGCTRL::inorder`.
- `mctrl`: An object of `SRT_MSGCTRL` type that contains extra parameters, including `ttl` and `inorder`.

The way this function works is determined by the mode set in options, and it has specific requirements:

1. In **file/stream mode**, the payload is byte-based. You are not required to know the size of the data, although they are only guaranteed to be received in the same byte order.

2. In **file/message mode**, the payload that you send using this function is a single message that you intend to be received as a whole. In other words, a single call to this function determines a message's boundaries.

3. In **live mode**, you are only allowed to send up to the length of `SRTO_PAYLOADSIZE`, which can't be larger than 1456 bytes (1316 default).

- Returns:

  - Size of the data sent, if successful. Note that in **file/stream mode** the returned size may be less than `len`, which means that it didn't send the whole contents of the buffer. You would need to call this function again with the rest of the buffer next time to send it completely. In both **file/message** and **live mode** the successful return is always equal to `len`
  - In case of error, `SRT_ERROR` (-1)

- Errors:

  - `SRT_ENOCONN`: Socket `u` used when the operation is not connected.
  - `SRT_ECONNLOST`: Socket `u` used for the operation has lost its connection.
  - `SRT_EINVALMSGAPI`: Incorrect API usage in **message mode**:
    - **live mode**: trying to send more bytes at once than `SRTO_PAYLOADSIZE`
  - `SRT_EINVALBUFFERAPI`: Incorrect API usage in **stream mode**:
    - Reserved for future use. The congestion controller object used for this mode doesn't use any restrictions on this call for now, but this may change in future.
  - `SRT_ELARGEMSG`: Message to be sent can't fit in the sending buffer (that is, it exceeds the current total space in the sending buffer in bytes). This means that the sender buffer is too small, or the application is trying to send a larger message than initially predicted.
  - `SRT_EASYNCSND`: There's no free space currently in the buffer to schedule the payload. This is only reported in non-blocking mode (`SRTO_SNDSYN` set to false); in blocking mode the call is blocked until enough free space in the sending buffer becomes available.
  - `SRT_ETIMEOUT`: The condition described above still persists and the timeout has passed. This is only reported in blocking mode when `SRTO_SNDTIMEO` is set to a value other than -1.
  - `SRT_EPEERERR`: This is reported only in the case where, as a stream is being received by a peer, the `srt_recvfile` function encounters an error during a write operation on a file. This is reported by a `UMSG_PEERERROR` message from the peer, and the agent sets the appropriate flag internally. This flag persists up to the moment when the connection is broken or closed.

srt_recv, srt_recvmsg, srt_recvmsg2

```
int srt_recv(SRTSOCKET u, char* buf, int len);
int srt_recvmsg(SRTSOCKET u, char* buf, int len);
int srt_recvmsg2(SRTSOCKET u, char *buf, int len, SRT_MSGCTRL *mctrl);
```

Extracts the payload waiting to be received. Note that `srt_recv` and `srt_recvmsg` are identical functions, two different names being kept for historical reasons. In the UDT predecessor the application was required to use either the `UDT::recv` version for **stream mode** and `UDT::recvmsg` for **message mode**. In SRT this distinction is resolved internally by the `SRTO_MESSAGEAPI` flag.

- `u`: Socket used to send. The socket must be connected for this operation.
- `buf`: Points to the buffer to which the payload is copied
- `len`: Size of the payload specified in `buf`
- `mctrl`: An object of `SRT_MSGCTRL` type that contains extra parameters

The way this function works is determined by the mode set in options, and it has specific requirements:

1. In **file/stream mode**, as many bytes as possible are retrieved, that is, only so many bytes that fit in the buffer and are currently available. Any data that is available but not extracted this time will be available next time.

2. In **file/message mode**, exactly one message is retrieved, with the boundaries defined at the moment of sending. If some parts of the messages are already retrieved, but not the whole message, nothing will be received (the function blocks or returns `SRT_EASYNCRCV`). If the message to be returned does not fit in the buffer, nothing will be received and the error is reported.

3. In **live mode**, the function behaves as in **file/message mode**, although the number of bytes retrieved will be at most the size of `SRTO_PAYLOADSIZE`. In this mode, however, with default settings of `SRTO_TSBPDMODE` and `SRTO_TLPKTDROP`, the message will be received only when its time to play has come, and until then it will be kept in the receiver buffer; also, when the time to play has come for a message that is next to the currently lost one, it will be delivered and the lost one dropped.

- Returns:

  - Size (>0) of the data received, if successful.
  - 0, if the connection has been closed
  - `SRT_ERROR` (-1) when an error occurs

- Errors:

  - `SRT_ENOCONN`: Socket `u` used for the operation is not connected.
  - `SRT_ECONNLOST`: Socket `u` used for the operation has lost connection (this is reported only if the connection was unexpectedly broken, not when it was closed by the foreign host).
  - `SRT_EINVALMSGAPI`: Incorrect API usage in **message mode**:
    - **live mode**: size of the buffer is less than `SRTO_PAYLOADSIZE`
  - `SRT_EINVALBUFFERAPI`: Incorrect API usage in **stream mode**:

        - Currently not in use. File congestion control used for **stream mode** does not restrict the parameters. **???**
    - SRT_ELARGEMSG: Message to be sent can't fit in the sending buffer (that is, it exceeds the current total space in the sending buffer in bytes). This means that the sender buffer is too small, or the application is trying to send a larger message than initially intended.
    - SRT_EASYNCRCV: There are no data currently waiting for delivery. This happens only in non-blocking mode (when SRTO_RCVSYN is set to false). In blocking mode the call is blocked until the data are ready. How this is defined, depends on the mode:
    - In **live mode** (with SRTO_TSBPDMODE on), at least one packet must be present in the receiver buffer and its time to play be in the past
    - In **file/message mode**, one full message must be available,
        - the next one waiting if there are no messages with inorder = false, or possibly the first message ready with inorder = false
    - In **file/stream mode**, it is expected to have at least one byte of data still not extracted
    - SRT_ETIMEOUT: The readiness condition described above is still not achieved and the timeout has passed. This is only reported in blocking mode when SRTO_RCVTIMEO is set to a value other than -1.

srt_sendfile, srt_recvfile

```
int64_t srt_sendfile(SRTSOCKET u, const char* path, int64_t* offset, int64_t size,
int block);
int64_t srt_recvfile(SRTSOCKET u, const char* path, int64_t* offset, int64_t size,
int block);
```

These are functions dedicated to sending and receiving a file. You need to call this function just once for the whole file, although you need to know the size of the file prior to sending and also define the size of a single block that should be internally retrieved and written into a file in a single step. This influences only the performance of the internal operations; from the application perspective you just have one call that exits only when the transmission is complete.

- u: Socket used for transmission. The socket must be connected.
- path: Path to the file that should be read or written.
- offset: Needed to pass or retrieve the offset used to read or write to a file
- size: Size of transfer (file size, if offset is at 0)
- block: Size of the single block to read at once before writing it to a file

The following values are recommended for the block parameter:

```
#define SRT_DEFAULT_SENDFILE_BLOCK 364000
#define SRT_DEFAULT_RECVFILE_BLOCK 7280000
```

You need to pass them to the srt_sendfile or srt_recvfile function if you don't know what value to chose.

- Returns:
  - Size (>0) of the transmitted data of a file. It may be less than `size`, if the size was greater than the free space in the buffer, in which case you have to send rest of the file next time.
  - -1 in case of error.
- Errors:
  - `SRT_ENOCONN`: Socket `u` used for the operation is not connected.
  - `SRT_ECONNLOST`: Socket `u` used for the operation has lost its connection.
  - `SRT_EINVALBUFFERAPI`: When socket has `SRTO_MESSAGEAPI` = true or `SRTO_TSBPDMODE` = true. (**BUG?**: Looxlike MESSAGEAPI isn't checked)
  - `SRT_EINVRDOFF`: There is a mistake in `offset` or `size` parameters, which should match the index availability and size of the bytes available since `offset` index. This is actually reported for `srt_sendfile` when the `seekg` or `tellg` operations resulted in error.
  - `SRT_EINVWROFF`: Like above, reported for `srt_recvfile` and `seekp`/`tellp`.
  - `SRT_ERDPERM`: The read from file operation has failed (`srt_sendfile`).
  - `SRT_EWRPERM`: The write to file operation has failed (`srt_recvfile`).

Diagnostics

General notes concerning the "getlasterror" diagnostic functions: when an API function ends up with error, this error information is stored in a thread-local storage. This means that you'll get the error of the operation that was last performed as long as you call this diagnostic function just after the failed function has returned. In any other situation the information provided by the diagnostic function is undefined.

srt_getlasterror

```
int srt_getlasterror(int* errno_loc);
```

Get the numeric code of the last error. Additionally, in the variable passed as `errno_loc` the system error value is returned, or 0 if there was no system error associated with the last error. The system error is:

- On POSIX systems, the value from `errno`
- On Windows, the result from `GetLastError()` call

srt_strerror

```
const char* srt_strerror(int code, int errnoval);
```

Returns a string message that represents a given SRT error code and possibly the `errno` value, if not 0.

NOTE: *This function isn't thread safe. It uses a static variable to hold the error description. There's no problem with using it in a multithreaded environment, as long as only one thread in the whole application calls this function at the moment*

srt_getlasterror_str

```
const char* srt_getlasterror_str(void);
```

Get the text message for the last error. It's a shortcut to calling first `srt_getlasterror` and then passing the returned value into `srt_strerror`. Note that, in contradiction to `srt_strerror`, this function is thread safe.

srt_clearlasterror

```
void srt_clearlasterror(void);
```

This function clears the last error. After this call, the `srt_getlasterror` will report a "successful" code.

srt_getrejectreason

```
enum SRT_REJECT_REASON srt_getrejectreason(SRTSOCKET sock);
```

This function shall be called after a connecting function (such as `srt_connect`) has returned an error, which's code was `SRT_ECONNREJ`. It allows to get a more detailed rejection reason. This function returns a numeric code, which can be translated into a message by `srt_rejectreason_str`. The following codes are currently reported:

SRT_REJ_UNKNOWN

A fallback value for cases when there was no connection rejected.

SRT_REJ_SYSTEM

One of system function reported a failure. Usually this means some system error or lack of system resources to complete the task.

SRT_REJ_PEER

The connection has been rejected by peer, but no further details are available. This usually means that the peer doesn't support rejection reason reporting.

SRT_REJ_RESOURCE

A problem with resource allocation (usually memory).

SRT_REJ_ROGUE

The data sent by one party to another cannot be properly interpreted. This should not happen during normal usage, unless it's a bug, or some weird events are happening on the network.

SRT_REJ_BACKLOG

The listener's backlog has exceeded (there are many other callers waiting for the opportunity of being connected and wait in the queue, which has reached its limit).

SRT_REJ_IPE

Internal Program Error. This should not happen during normal usage and it usually means a bug in the software (although this can be reported by both local and foreign host).

SRT_REJ_CLOSE

The listener socket was able to receive your request, but at this moment it is being closed. It's likely that your next attempt will result with timeout.

SRT_REJ_VERSION

Any party of the connection has set up minimum version that is required for that connection, and the other party didn't satisfy this requirement.

SRT_REJ_RDVCOOKIE

Rendezvous cookie collision. This normally should never happen, or the probability that this will really happen is negligible. However this can be also a result of a misconfiguration that you are trying to make a rendezvous connection where both parties try to bind to the same IP address, or both are local addresses of the same host - in which case the sent handshake packets are returning to the same host as if they were sent by the peer, who is this party itself. When this happens, this reject reason will be reported by every attempt.

SRT_REJ_BADSECRET

Both parties have defined a passprhase for connection and they differ.

SRT_REJ_UNSECURE

Only one connection party has set up a password. See also `SRTO_STRICTENC` flag in API.md.

SRT_REJ_MESSAGEAPI

The value for `SRTO_MESSAGEAPI` flag is different on both connection parties.

SRT_REJ_CONGESTION

The `SRTO_CONGESTION` option has been set up differently on both connection parties.

SRT_REJ_FILTER

The `SRTO_FILTER` option has been set differently on both connection parties (NOTE: this flag may not exist yet in this version).

srt_rejectreason_str

```
const char* srt_rejectreason_str(enum SRT_REJECT_REASON id);
```

Returns a constant string for the reason of the connection rejected, as per given code id. Alternatively you can use the `srt_rejectreason_msg` array. This function additionally handles the case for unknown id by reporting `SRT_REJ_UNKNOWN` in such case.

Performance tracking

General note concerning sequence numbers used in SRT: they are 32-bit "circular numbers" with the most significant bit not included, so for example 0x7FFFFFFF shifted by 3 forward becomes 2. As far as any comparison is concerned, it can be only spoken about a "distance" rather than difference, which is an integer value expressing an offset to be added to one sequence in order to get the second one. This distance is only valid as long as the threshold value isn't exceeded, so it's stated that all sequence numbers that are anywhere taken into account were systematically updated and they are kept in the range between 0 and half of the maximum 0x7FFFFFFF. Hence the distance counting procedure always assumes that the sequence number are in the required range already, so for a numbers like 0x7FFFFFF0 and 0x10, for which the "numeric difference" would be 0x7FFFFFE0, the "distance" is 0x20.

srt_bstats, srt_bistats

```
// perfmon with Byte counters for better bitrate estimation.
int srt_bstats(SRTSOCKET u, SRT_TRACEBSTATS * perf, int clear);

// permon with Byte counters and instantaneous stats instead of moving averages
for Snd/Rcvbuffer sizes.
int srt_bistats(SRTSOCKET u, SRT_TRACEBSTATS * perf, int clear, int
instantaneous);
```

Reports the current statistics

- `u`: Socket from which to get statistics
- `perf`: Pointer to an object to be written with the statistics
- `clear`: 1 if the statistics should be cleared after retrieval
- `instantaneous`: 1 if the statistics should use instant data, not moving averages

`SRT_TRACEBSTATS` is an alias to `struct CBytePerfMon`. Most of the fields are reasonably well described in the header file comments. Here are descriptions of some less obvious fields in this structure (instant measurements):

- `usPktSndPeriod`: This is the minimum time (sending period) that must be kept between two packets sent consecutively over the link used by this socket. Note that sockets sharing one outgoing port use the same underlying UDP socket and therefore the same link and the same sender queue. `usPktSndPeriod` is the inversion of the maximum sending speed. It isn't the EXACT time interval between two consecutive sendings because in the case where the time spent by the application between two consecutive sendings exceeds `usPktSndPeriod`, the next packet will be sent immediately. The extra "wasted" time will be accounted for at the next sending.

- `pktFlowWindow`: The "flow window" in packets. It is the amount of free space on the peer receiver, stating that this socket represents the sender. When this value drops to zero, the next packet sent will be dropped by the receiver without processing. In **file mode** this may cause a slowdown of sending in order to wait until the receiver makes more space available, after it eventually extracts the packets waiting in its receiver buffer; in **live mode** the receiver buffer contents should normally occupy not more than half of the buffer size (default 8192). If `pktFlowWindow` value is less than that and becomes even less in the next reports, it means that the receiver application on the peer side cannot process the incoming stream fast enough and this may lead do a dropped connection.

- `pktCongestionWindow`: The "congestion window" in packets. In **file mode** this value starts at 16 and is increased with every number of reported acknowledged packets, and then is also updated based on the receiver-reported delivery rate. It represents the maximum number of packets that can be safely sent now without causing congestion. The higher this value, the faster the packets can be sent. In **live mode** this field is not used.

- `pktFlightSize`: The number of packets in flight. This is the distance between the packet sequence number that was last reported by an ACK message and the sequence number of the packet just sent (at the moment when the statistics are being read).

NOTE: ACKs are received periodically, so this value is most accurate just after receiving an ACK and becomes a little exaggerated over time until the next ACK arrives. This is because with a new packet sent and the sent sequence increased the ACK number stays the same for a moment, which increases this value, but the exact number of packets arrived since the last ACK report is unknown. Possibly a new statistical data can be added which holds only the distance between the ACK sequence and the sent sequence at the moment when ACK arrives and isn't updated until the next ACK arrives. The difference between this value and `pktFlightSize` would show then the number of packets whose fate is unknown at the moment.

- `msRTT`: The RTT (Round-Trip time) is the sum of two STT (Single-Trip time) values, one from agent to peer, and one from peer to agent. Note that **the measurement method is different than on TCP**; SRT measures only the "reverse RTT", that is, the time measured at the receiver between sending a `UMSG_ACK` message until receiving the sender-responded `UMSG_ACKACK` message (with the same journal). This happens to be a little different to the "forward RTT" as measured in TCP, which is the time between sending a data packet of a particular sequence number and receiving `UMSG_ACK` with a sequence number that is later by 1. Forward RTT isn't being measured or reported in SRT, although some research works have shown that these values, even though shuold be the same, happen to differ, that is, "reverse RTT" seems to be more optimistic.

- `mbpsBandwidth`: The bandwidth in Mb/s. The bandwidth is measured at the receiver, which sends back a running average calculation to the sender with the ACK message.

- `byteAvailSndBuf`: The number of bytes available in the sender buffer. This value decreases with data scheduled for sending by the application, and increases with every ACK received from the receiver, after the packets are sent over the UDP link.

- `byteAvailRcvBuf`: The number of bytes available in the receiver buffer. This value increases after the application extracts the data from the socket (uses one of `srt_recv*` functions) and decreases with every packet received from the sender over the UDP link.

- `mbpsMaxBW`: The maximum bandwidth in Mb/s. Usually this is the setting from the `SRTO_MAXBW` option, which may include the value 0 (unlimited). Under certain conditions a nonzero value might be be provided by the appropriate congestion control module, although none of the built-in congestion control modules currently uses it.

- `byteMSS`: Same as a value from `SRTO_MSS` option, "Message Segment Size". It's the size of the MTU unit (size of the UDP packet used for transport, including all possible headers, that is Ethernet, IP and UDP), default 1500.

- `pktSndBuf`: The number of packets in the send buffer that are already scheduled for sending or even possibly sent, but not yet acknowledged.

- `byteSndBuf`: Same as `pktSndBuf`, in bytes.

- `msSndBuf`: Same as `pktSndBuf`, but expressed as a time interval between the oldest and the latest packet scheduled for sending.

- `msSndTsbPdDelay`: If `SRTO_TSBPDMODE` is on (default for **live mode**), it returns the value of `SRTO_PEERLATENCY`, otherwise 0.

- `pktRcvBuf`: Number of packets in the receiver buffer. Note that in **live mode** (with `SRTO_TSBPDMODE` turned on, default) some packets must stay in the buffer and will not be signed off to the application until the "time to play" comes. In **file mode** (both stream and message) it means that all that is above 0 can (and shall) be read right now.

- `byteRcvBuf`: Like `pktRcvBuf`, in bytes.

- `msRcvBuf`: Time interval between the first and last available packets in the receiver buffer. Note that this range includes all packets regardless of whether they are ready to play or not (regarding the **live mode**)..

- `msRcvTsbPdDelay`: If `SRTO_TSBPDMODE` is on (default for **live mode**), it returns the value of `SRTO_RCVLATENCY`; otherwise 0.

Asynchronous operations (epoll)

The epoll system is currently the only method for using multiple sockets in one thread with having the blocking operation moved to epoll waiting so that it can block on multiple sockets at once. That is, instead of blocking a single reading or writing operation, as it's in blocking mode, it blocks until at least one of the sockets subscribed for a single waiting call in given operation mode is ready to do this operation without blocking. It's usually combined with setting the nonblocking mode on a socket, which in SRT is set separately for reading and writing (`SRTO_RCVSYN` and `SRTO_SNDSYN` respectively) in order to ensure that in case of some internal error in the application (or even possibly a bug in SRT that has reported a spurious readiness report) the operation will end up with error rather than cause blocking, which would be more dangerous for the application in this case (`SRT_EASYNCRCV` and `SRT_EASYNCRCV` respectively).

The epoll system, similar to the one on Linux, relies on `eid` objects managed internally in SRT, which can be subscribed to particular sockets and the readiness status of particular operations. The `srt_epoll_wait` function can then be used to block until any readiness status in the whole `eid` is set.

srt_epoll_create

```
int srt_epoll_create(void);
```

Creates a new epoll container.

- Returns:
    - valid EID on success
    - -1 on failure
- Errors:
    - `SRT_ECONNSETUP`: System operation failed. This is on systems that use a special method for the system part of epoll and therefore associated resources, like epoll on Linux.

srt_epoll_add_usock, srt_epoll_add_ssock, srt_epoll_update_usock, srt_epoll_update_ssock

```
int srt_epoll_add_usock(int eid, SRTSOCKET u, const int* events);
int srt_epoll_add_ssock(int eid, SYSSOCKET s, const int* events);
int srt_epoll_update_usock(int eid, SRTSOCKET u, const int* events);
int srt_epoll_update_ssock(int eid, SYSSOCKET s, const int* events);
```

Adds a socket to a container, or updates an existing socket subscription.

The `_usock` suffix refers to a user socket (SRT socket). The `_ssock` suffix refers to a system socket.

The `_add_` functions add new sockets. The `_update_` functions act on a socket that is in the container already and just allow changes in the subscription details. For example, if you have already subscribed a socket with `SRT_EPOLL_OUT` to wait until it's connected, to change it into poll for read-readiness, you use this function on that same socket with a variable set to `SRT_EPOLL_IN`. This will not only change the event type which is polled on the socket, but also remove any readiness status for flags that are no longer set. It is discouraged to perform socket removal and adding back (instead of using `_update_`) because this way you may miss an event that could happen in a short moment between these two calls.

- `eid`: epoll container id
- `u`: SRT socket
- `s`: system socket
- `events`: points to a variable set to epoll flags, or NULL if you want to subscribe a socket for all possible events
- Returns:
    - 0 if successful, otherwise -1
- Errors:
    - `SRT_EINVPOLLID`: `eid` designates no valid EID object

BUG?: for `add_ssock` the system error results in an empty `CUDTException()` call which actually results in `SRT_SUCCESS`. For cases like that the `SRT_ECONNSETUP` code is predicted.

srt_epoll_remove_usock, srt_epoll_remove_ssock

```
int srt_epoll_remove_usock(int eid, SRTSOCKET u);
int srt_epoll_remove_ssock(int eid, SYSSOCKET s);
```

Removes a specified socket from an epoll container and clears all readiness states recorded for that socket.

The `_usock` suffix refers to a user socket (SRT socket). The `_ssock` suffix refers to a system socket.

- Returns:
  - 0 if successful, otherwise -1
- Errors:
  - `SRT_EINVPOLLID`: `eid` designates no valid EID object

srt_epoll_wait

```
int srt_epoll_wait(int eid, SRTSOCKET* readfds, int* rnum, SRTSOCKET* writefds,
int* wnum, int64_t msTimeOut,
                        SYSSOCKET* lrfds, int* lrnum, SYSSOCKET* lwfds, int*
lwnum);
```

Blocks the call until any readiness state occurs in the epoll container. Mind that the readiness states reported in epoll are **permanent, not edge-triggered**.

Readiness can be on a socket in the container for the event type as per subscription. The first readiness state causes this function to exit, but all ready sockets are reported. This function blocks until the timeout. If timeout is 0, it exits immediately after checking. If timeout is -1, it blocks indefinitely until a readiness state occurs.

- `eid`: epoll container
- `readfds` and `rnum`: A pointer and length of an array to write SRT sockets that are read-ready
- `writefds` and `wnum`: A pointer and length of an array to write SRT sockets that are write-ready
- `msTimeOut`: Timeout specified in milliseconds, or special values (0 or -1)
- `lwfds` and `lwnum`:A pointer and length of an array to write system sockets that are read-ready
- `lwfds` and `lwnum`:A pointer and length of an array to write system sockets that are write-ready

Note that there is no space here to report sockets for which it's already known that the operation will end up with error (athough such a state is known internally). If an error occurred on a socket then that socket is reported in both read-ready and write-ready arrays, regardless of what event types it was subscribed for. Usually then you subscribe given socket for only read readiness, for example (`SRT_EPOLL_IN`), but pass both arrays for read and write readiness. This socket will not be reported in the write readiness array even if it's write ready (because this isn't what it was subscribed for), but it will be reported there, if the next operation on this socket is about to be erroneous. On such sockets you can still perform an operation, just you should expect that it will always report and error. On the other hand that's the only way to know what kind of error has occurred on the socket.

- Returns:
    - The number (>0) of ready sockets, of whatever kind (if any)
    - -1 in case of error
- Errors:
    - `SRT_EINVPOLLID`: `eid` designates no valid EID object
    - `SRT_ETIMEOUT`: Up to `msTimeOut` no sockets subscribed in `eid` were ready. This is reported only if `msTimeOut` was >=0, otherwise the function waits indefinitely.

srt_epoll_release

```
int srt_epoll_release(int eid);
```

Deletes the epoll container.

- Returns:
    - The number (>0) of ready sockets, of whatever kind (if any)
    - -1 in case of error
- Errors:
    - `SRT_EINVPOLLID`: `eid` designates no valid EID object

Logging control

SRT has a widely used system of logs, as this is usually the only way to determine how the internals are working, without changing the rules by the act of tracing. Logs are split into levels (5 levels out of those defined by syslog are in use) and additional filtering is possible on FA (functional area). By default only up to the *Note* log level are displayed and from all FAs.

Logging can only be manipulated globally, with no regard to a specific socket. This is because lots of operations in SRT are not dedicated to any particular socket, and some are shared between sockets.

srt_setloglevel

```
void srt_setloglevel(int ll);
```

Sets the minimum severity for logging. A particular log entry is displayed only if it has a severity greater than or equal to the minimum. Setting this value to `LOG_DEBUG` turns on all levels.

The constants for this value are those from `<sys/syslog.h>` (for Windows, refer to `common/win/syslog_defs.h`). The only meaningful are:

- `LOG_DEBUG`: Highly detailed and very frequent messages
- `LOG_NOTICE`: Occasionally displayed information
- `LOG_WARNING`: Unusual behavior
- `LOG_ERR`: Abnormal behavior
- `LOG_CRIT`: Error that makes the current socket unusable

srt_addlogfa, srt_dellogfa, srt_resetlogfa

```
void srt_addlogfa(int fa);
void srt_dellogfa(int fa);
void srt_resetlogfa(const int* fara, size_t fara_size);
```

A functional area (FA) is an additional filtering mechanism for logging. You can set up logging to display logs only from selected FAs. The list of FAs is collected in `srt.h` file, as identified by the `SRT_LOGFA_` prefix. They are not enumerated here because they may be changed very often.

All FAs are turned on by default, except potentially dangerous ones (such as `SRT_LOGFA_HAICRYPT`). The reaons is that they may display either some security information that shall remain in the memory only (so, only if strictly required for the development), or some duplicated information (so you may want to turn this FA on, while turning off the others).

srt_setloghandler

```
void srt_setloghandler(void* opaque, SRT_LOG_HANDLER_FN* handler);
typedef void SRT_LOG_HANDLER_FN(void* opaque, int level, const char* file, int
line, const char* area, const char* message);
```

By default logs are printed to standard error stream. This function replaces the sending to a stream with a handler function that will receive them.

srt_setlogflags

```
void srt_setlogflags(int flags);
```

When you set a log handler with `srt_setloghandler`, you may also want to configure which parts of the log information you do not wish to be passed in the log line (the `message` parameter). A user's logging facility may, for example, not wish to get the current time or log level marker, as it will provide this information on its own.

The following flags are available, as collected in `logging_api.h` public header:

- `SRT_LOGF_DISABLE_TIME`: Do not provide the time in the header

- `SRT_LOGF_DISABLE_THREADNAME`: Do not provide the thread name in the header
- `SRT_LOGF_DISABLE_SEVERITY`: Do not provide severity information in the header
- `SRT_LOGF_DISABLE_EOL`: Do not add the end-of-line character to the log line

RETURN TO TOP OF PAGE