Documentation of the SCTP-Implementation
Release: sctplib-1.0

Andreas Jungmaier
ajung@exp-math.uni-essen.de
Michael Tüxen
Michael.Tuexen@icn.siemens.de

February 22, 2001

1 Nomenclature

Throughout this document,

- function names are written as **func_name**(),
- function parameters as `parameter`,
- data types as **typename**, and
- file names as *name_of_file*.

2 General Concepts

The sctplib-1.0 library release is the product of a cooperation between Siemens AG (ICN), Munich, Germany and the Computer Networking Technology Group at the IEM of the University of Essen, Germany. It has been developed since late 1999 and has been planned to become a fairly complete prototype implementation of the SCTP protocol as described in [1].

The API of the function library was modeled after section 10 of the RFC 2960 [1], and most parameters should be self-explanatory to the user familiar with this document. In addition to the interface functions between an Upper Layer Protocol (ULP) and an SCTP instance, the library also provides a number of helper functions that can be used to manage callback function routines to make them execute at a certain point of time (i.e. timer-based) or open and bind *local* UDP sockets on a configurable port, which may then be used for an asynchronous interprocess communication.

All of these functions may be made use of by simply linking the static `libsctp`-library to an application, and including the file *sctp.h*. As SCTP operates on top of IP, we chose to open a raw socket to catch all incoming packets with the ip protocol id byte set to 132 (=SCTP). For this, it is absolutely necessary that an application using the library functions have privileges to do so. On most Unix machines that will mean, this application has to be run by root (i.e. be made setuid 0).

An application making use of the `libsctp`-library can expect to be able to manage a large number of associations (e.g. as a server, all bound to one port) as well as handle several *SCTP instances*, which may work on several different ports. Ports of incoming packets are checked whether they belong to an already existing association or may start a new one.

The concept of the library is the following: After registration of a first *SCTP instance*, sockets are openend and an application may register file descriptors, that asynchronously trigger execution of callback functions or timer events. The function callbacks for these routines are passed in a **struct ULPcallback**.

Then the application either calls the possibly blocking function **sctp_event_loop**() or the nonblocking function **sctp_get_events**(). While calling the former, it reacts to a previously scheduled timer or any file descriptor events (by executing the registered callback functions). In case a timer is scheduled at a very late point in time, and no events happen on registered file descriptors (e.g. sockets), the program will sleep (because the system call **poll**() is used ! In this case, the control flow is handled by the library, and the user **must** register appropriate callbacks for events and timers before handing control over to the **sctp_event_loop**() function. The proper use of the **sctp_event_loop**() is explained in some simple example programs explained is section 6.

On the other hand, there is also the function **sctp_get_events**() which may be used to check for timer or file descriptor events. If there are none, it returns immediatly. While scheduling CPU intensive functions to run, an application should call this function in between in order to treat SCTP events in between this

2.1 Notes

Note that the `libsctp`-library depends on a number of software packets such as glib-1.2 (for portable definition of types, list functions etc.). Since we started using these packets late during the project's development cycle, its application is sometimes not very consequent. We continue working on that, though, so things should clean up a bit during later releases. For one, removal of the *dll_main.c/.h* files is planned.

3 Features of the Implementation

The implementation is able to establish an association according to RFC2960 and its interoperablility has been tested with implementations of about 25 companies. It features a complete API according to section 10 of RFC 2960 [1]. Naturally it supports the SCTP multihoming feature, and is able to handle all-IPv4 as well as all-IPv6 or mixed IPv4/IPv6 associations. The association may be established using a Cookie-structure that is secured by a secret key and a secure MD5 hash function message authentication code.

The data path implements the flow control features as prescribed in the RFC, and supports the delayed acknowledgement feature, delay of which may be configured, optionally. Using the **sctp_setAssociationParams**() and **sctp_setInstanceParams**() functions, a wide range of protocol parameters may optionally be configured for each association.

There can be several SCTP instances per host, provided they have a set of disjoint addresses. The implementation also supports *reassembly* of fragmented chunks, and delivery of unordered chunks, as well as ordered chunks.

4 What is Missing

- Reception and creation of Error Chunks is implemented, but only as hooks that do not actually do anything.
- Handling of ICMP messages is not currently being done. So we do NOT do MTU path discovery.
- Segmentation is NOT done. This requires remodeling the StreamEngine module. That is for a later release !
- Some API functions are in the API, but currently do not do anything !
- Probably there is some more.Additions, comments, bug-reports and bug-fixes are always welcome !

5 The API

5.1 Constants

When the status of a path changes, the following values may be assumed:

```
#define PATH_OK              0
#define PATH_UNREACHABLE     1
```

For setting the heartbeat (see section 5.4.9, **sctp_changeHeartBeat**()), these constants may be used:

```
#define HEARTBEAT_ON         1
#define HEARTBEAT_OFF        0
```

When calling **sctp_send**() (see section 5.4.5), the following constants may be used to select, how chunks are sent (i.e. for delivery with or without stream reordering, one may set:

```
#define UNORDERED_DELIVERY      1
#define ORDERED_DELIVERY        0
```

To prevent messages from being bundled, the following constants may be passed as parameters to **sctp_send**():

```
#define BUNDLING_ENABLED     0
#define BUNDLING_DISABLED    1
```

Two more constants may be needed frequently for **sctp_send**(),

1. for path selection, if the primary path is to be taken (which is the default):

   ```
   #define USE_PRIMARY         -1
   ```

2. this is used to have chunks have an unlimited lifetime:

   ```
   #define INFINITE_LIFETIME   0
   ```

5.2 Type Definitions

The main include file of *sctp.h* defines a number of types that are used throughout the library as well as in some of the interface functions. They will be explained in the subsequent sections.

5.2.1 struct ULPcallback

This is the structure containing pointers to functions (used as callbacks for events that may occur and that the ULP needs to be notified of), which are all explained in detail in section 5.5.
This structure is usually initialized early in the program, and then passed to the function **sctp_registerSCTP_instance**() (see section 5.4.1), which in turn registers the appropriate functions from this structure for the corresponding event. Not all of these functions are currently called, but appropriate callback routines should be registered anyway, because this will change in later releases.
Definition:

```
struct ULPcallback
{
    void (*dataArriveNotif) (unsigned int, unsigned int, unsigned int,
                             unsigned int, unsigned int,   void*);
    void (*sentFailureNotif) (unsigned int, unsigned char *, unsigned int,
                              unsigned int *, void*);
    void (*networkStatusChangeNotif) (unsigned int, short, unsigned short, void*);
    void* (*communicationUpNotif) (unsigned int, unsigned short, unsigned char* [][],
                                   int, unsigned short, unsigned short, void*);
    void (*communicationLostNotif) (unsigned int, unsigned short, void*);
    void (*communicationErrorNotif) (unsigned int, unsigned short, void*);
    void (*RestartNotif) (unsigned int, void*);
    void (*ShutdownCompleteNotif) (unsigned int, void*);
};
```

5.2.2 InstanceParams

This struct contains variables that may be set per SCTP instance, and can subsequently set or got after an instance has been registered. All associations that will be created from this instance will then inherit the parameters.
Definition:

```
typedef struct InstanceParameters {
    /// the initial round trip timeout
    unsigned int RTO_initial;
    /// the lifetime of a cookie
    unsigned int validCookieLife;
    /// maximum retransmissions per association
    unsigned int assocMaxRetransmits;
    /// maximum retransmissions per path
    unsigned int pathMaxRetransmits;
    /// maximum initial retransmissions
    unsigned int maxInitRetransmits;
    /// from recvcontrol : my receiver window
    unsigned int my_rwnd;
    /// recvcontrol: delay for delayed ACK in msecs
    unsigned int delay;
    /// per path ? per instance ? for the IP type of service field.
    unsigned char ip_tos;
    /// currently unused, will limit the size of the send queue later
    unsigned int max_sendQueue;
    /// currently unused, may limit the size of the receive queue later.
    unsigned int max_recvQueue;
} InstanceParams;
```

5.2.3 AssocParams

This struct contains variables that may be set per SCTP association, and can subsequently set or got after a COMMUNICATION_UP has been received. Among others it has the following values:

`RTO_initial` initial round trip time.

`validCookieLife` cookie life time.

`assocMaxRetransmits` maximum number of retransmission before the peer is considered unreachable.

`maxInitRetransmits` maximum number of retransmission before a destination address is considered unreachable.

`maxInitRetransmits` maximum number of retransmissions of init and cookie messages.

Definition:

```
typedef struct AssociationParameters {
    unsigned int RTO_initial;
    ///
    unsigned int validCookieLife;
    ///
    unsigned int assocMaxRetransmits;
    ///
    unsigned int pathMaxRetransmits;
    ///
    unsigned int maxInitRetransmits;
    /// smoothed round trip time, per path
    unsigned int srtt;
    /// round trip time variation, per path
    unsigned int rttvar;
    /// defines the rate at which heartbeats are sent
    unsigned int heartbeatIntervall;
    ///  flowcontrol per path
    unsigned int cwnd;
```

```
    ///  flowcontrol, per path
    unsigned int cwnd2;
    ///  flowcontrol, per path
    unsigned int partial_bytes_acked;
    ///  flowcontrol, per path
    unsigned int ssthresh;
    ///  flowcontrol, per path
    unsigned int outstanding_bytes_per_address;
    ///  flowcontrol, per path
    unsigned int mtu;
    /// per path ? per instance ? for the IP type of service field.
    unsigned char ip_tos;
} AssocParams;
```

5.2.4 AssociationStatus

This struct contains the data returned to the ULP after calling the status primitive (function **sctp_status**()).

Definition:

```
typedef struct SCTP_Status
{
    unsigned short association_state;
    unsigned short number_of_addresses;
    unsigned char *primaryDestinationAddress[];
    unsigned short outstreams;
    unsigned short instreams;
    unsigned short primaryAddressIndex;
    unsigned char *destinationAddresses[][];
    short *path_states;
    unsigned int *SRTT;
    unsigned int *RTO;
    unsigned int currentReceiverWindowSize;
    unsigned int  *currentCongestionWindowSize;
    unsigned int noOfChunksInSendQueue;
    unsigned int noOfChunksInRetransmissionQueue;
    unsigned int noOfChunksInReceptionQueue;
}AssociationStatus;
```

Pointers point to arrays that are indexed by the number of destination addresses. So if there is an association with three destination addresses, and a variable is declared as `AssociationStatus astatus*`, a call to **sctp_status**() yields values for all of these destination addresses. A single value may then be accessed as `astatus->SRTT[1]`, for example. The arrays are allocated by the sctp library, and need to be freed by the user application for now. We might later implement a function that frees that memory, too.

The values contained in this structure give most of the important data about the current association:

association_state returns an unsigned integer representing the state of the association. The states have been defined as:

- CLOSED = 0
- COOKIE_WAIT = 1
- COOKIE_ECHOED = 2
- ESTABLISHED = 3
- SHUTDOWNPENDING = 4
- SHUTDOWNRECEIVED = 5
- SHUTDOWNSENT = 6
- SHUTDOWNACKSENT = 7

number_of_addresses returns the number of destination addresses this association has, i.e. the number of possible paths.

destinationAddresses is a pointer to an array of zero-terminated strings containing the destination addresses.

path_states points to an array of values, that are either 0 (=`PM_ACTIVE`) or 1 (=`PM_INACTIVE`) and thus indicate the current state of all paths.

5.2.5 Socket_callback_function

Definition:

```
typedef void (*Socket_callback_function) (int, unsigned char *, int,
                                          unsigned char* [], int);
```

A callback function type used for registering callbacks for socket events (for events POLLIN or POLLPRI). The parameters that are passed by the sctp library, when a socket has been active are (in this order):

int socket file descriptor of the socket where a datagram was received

unsigned char* pointer to the data

int length of datagram

unsigned char* pointer to source address string (from which originated the data)

int length of this source address

5.2.6 Timer_expires_callback

Definition:

```
typedef void (*Timer_expires_callback) (unsigned int, void *, void *);
```

Defines the callback function that is called when an timer expires. Parameters:

unsigned int ID of timer

void* pointer to param1

void* pointer to param2

param1 and param2 are pointers to data that are returned to the callback function, when the timer expires. These must still exist at that time and point to valid data !

5.3 Helper Functions

The `libsctp`-library contains a few functions that influence the general control flow of a program using these, and are needed to create an application that works as expected. The important functions needed in *any* program using the `libsctp`-library are the functions **sctp_event_loop**() and **sctp_get_events**()

5.3.1 sctp_event_loop()

Basically this is a wrapper to a **poll**()-system call (if such a syscall exists on the used OS). The function waits until either of one events occurrs:

1. the time for which a timer event was scheduled has passed by, so the callback function belonging to that (previously registered) event is executed.
2. one of the sockets that had previously been registered with the function **sctp_register_udp_callback**() or the raw socket that waits for incoming SCTP packets has encountered a read event, so there is data available. An appropriate function is called to treat that event.

The control flow of the program is given to the callback function which may in turn register new timer events. The function returns -1 if an error occurrs, 0 if a timeout has occurred, or else the number of file descriptor events that have been treated.
Definition:

```
int sctp_event_loop();
```

5.3.2 sctp_get_events()

Basically this is a wrapper to a **poll**()-system call (if such a syscall exists on the used OS). The function checks whether a timer event needs to be handled and does so, if the time has arisen. The it checks via the **poll**()-system call whether events on the socket need to be handled, and does so, if there are any.

The function then returns after these events have been handled. It returns the number of events handled, 0 if none had to be handled, -1 if an error has occurred.

Definition:

```
int sctp_get_events();
```

5.3.3 sctp_register_udp_callback()

This function is supposed to open and bind a UDP socket listening on a port to incoming UDP pakets from a local IP address. After that it registers a callback function that is called, when UDP data arrives for that local address. The function takes the following parameters:

`me` pointer to local address string (zero-terminated)

`my_port` the UDP port to bind (locally)

`scf` callback funtion that is called when data has arrived

It returns a new UDP socket file descriptor, or -1 if error ocurred.

Definition:

```
int sctp_register_udp_callback(unsigned char* me[],
                               unsigned short my_port,
                               Socket_callback_function scf);
```

5.3.4 sctp_register_stdin_callback()

This function is supposed to register a callback function for catching input from the Unix STDIN file descriptor. We expect this to be useful in test programs mainly, so it is provided here for convenience.

`scf` callback funtion that is called when data has arrived

It returns 0, or -1 if an error ocurred.

Definition:

```
int sctp_register_stdin_callback(Socket_callback_function scf);
```

5.3.5 sctp_start_timer()

This function adds a callback that is to be called some time from now. It realizes the timer (in an ordered list). The function takes the following parameters:

`milliseconds` action is to be started in milliseconds ms from now

`timer_cb` pointer to a function to be executed, when timer expires

`param1` pointer to be returned to the caller when timer expires

`param2` pointer to be returned to the caller when timer expires

The function returns a timer ID value, that can be used to cancel or restart this timer.**NOTE:** the pointers `param1` and `param2` exist to point to data useful for the function callback. They need to point to data that is still valid at the time the callback is activated. Do not pass pointers to temporary objects !

Definition:

```
unsigned int sctp_start_timer(unsigned int milliseconds, Timer_expires_callback
                              timer_cb, void *param1, void *param2);
```

5.3.6 sctp_stop_timer()

This function stops a previously started timer. The function takes the following parameter:

`tid` timer-id of timer to be removed

The function returns 0 on success, 1 if tid not in the list, -1 on error.
Definition:

```
int sctp_stop_timer(unsigned int tid);
```

5.3.7 sctp_restart_timer()

Restarts a timer that is currently running. The function takes the following parameters:

`timer_id` the timer id returned by start_timer

`milliseconds` action is to be taken in milliseconds ms from now

The function returns a new timer id, zero when there is an error (i.e. timer was not running). The action basically stop the old timer, and sets a new timer. So it is there for convenience. You will have a different timer ID after calling this function !
Definition:

```
unsigned int sctp_restart_timer(unsigned int timer_id, unsigned int milliseconds);
```

5.3.8 sctp_gettime()

This function is a convenience wrapper to the standard Unix **gettimeofday**()-function, but sets only the struct timeval. It will return 0 for success, or -1 for failure (in which case errno is set appropriately).
Definition:

```
int sctp_gettime(struct timeval *tv);
```

5.4 ULP-to-SCTP

5.4.1 sctp_registerSCTP_instance()

sctp_registerSCTP_instance() is called to initialize one SCTP instance. On the very first call, it will open a raw socket for capturing SCTP packets and also the necessary ICMP events from the network. An application may register several instances with different callback functions, but there should not be several instances with the same port. If the port is set to zero, the instance will not be able to receive any datagrams, unless the **sctp_associate**() function is called. So it does not make sense to specify 0 as port number, unless we have a client. A server needs a port number greater than 0 here ! A client may have the `libsctp`-library choose a free source port by specifying zero here.
The function takes the following parameters:

`port` port of this SCTP instance

`noOfLocalAddresses` number of local addresses

`localAddressList` pointer to an array of local address-strings (zero-terminated)

`ULPcallbackFunctions` call back functions for primitives passed to ULP

The function returns the instance name of this new SCTP instance.
Definition:

```
unsigned short sctp_registerSCTP_instance(unsigned short port,
                          unsigned short noOfInStreams,
                          unsigned short noOfOutStreams,
                          unsigned int noOfLocalAddresses,
                          unsigned char* localAddressList[][],
                          struct ULPcallback ULPcallbackFunctions)
```

5.4.2 sctp_associate()

This function is called to set up an association. It triggers sending of an INIT chunk to a server, and when the association gets established, the `Communication Up`-notification is called. The ULP must specify the SCTP instance which this association belongs to. The function takes the following parameters:

`SCTP_InstanceName` the SCTP instance this association belongs to. If the local port of this SCTP instance is zero, we will get a free port number assigned, else we will use the one specified in the call to **sctp_registerSCTP_instance**().

`noOfOutStreams` number of output streams the ULP would like to have.

`destinationAddress` destination address string (zero-terminated) containing the address to which the INIT will be sent.

`destinationPort` destination port

The function returns the association ID of this association (identical with local tag), or 0 in case of failures.

Definition:

```
unsigned int sctp_associate(unsigned short SCTP_InstanceName,
                            unsigned short noOfOutStreams,
                            unsigned char *destinationAddress[],
                            unsigned short destinationPort)
```

5.4.3 sctp_shutdown()

sctp_shutdown() initiates the shutdown of the specified association. Basically triggers sending of a SHUTDOWN chunk. After the shutdown procedure is completed, the `Shutdown Complete`-Notification is called ! The function takes the following parameters:

`associationID` the ID of the addressed association.

The function returns 0 for success, 1 for error (assoc. does not exist).
Definition:

```
int sctp_shutdown(unsigned int associationID)
```

5.4.4 sctp_abort()

sctp_abort() initiates the abort of the specified association. The function takes the following parameters:

`associationID` the ID of the addressed association.

The function returns 0 for success, 1 for error (assoc. does not exist).
Definition:

```
int sctp_abort(unsigned int associationID)
```

5.4.5 sctp_send()

sctp_send() is used by the ULP to send data as data chunks. There are quite a few parameters that can be or must be passed along: The function takes the following parameters:

`associationID` the ID of the addressed association.

`streamID` identifies the stream on which the chunk is sent.

`buffer` chunk data.

`length` length of chunk data.

`protocolId` the payload protocol identifier

`destAddressIndex` index of destination address, if different from primary path. Primary Path is taken, if the constant `USE_PRIMARY` is used here.

`context` ULP context for this data chunk, to be returned in case of errors.

`lifetime` maximum time of chunk in send queue. Use constant for `INFINITE_LIFETIME`.

`unorderedDeliver` the chunk is delivered to peer ULP without resequencing. Use constants `ORDERED_DELIVERY` or `UNORDERED_DELIVERY`

`dontBundle` if true, chunk must not be bundled with other data chunks. Use constants `BUNDLING_ENABLED` or `BUNDLING_DISABLED`.

The function returns an error code: -1 for send error, 1 for association error, 0 if successful.
Definition:

```
int sctp_sendChunk(unsigned int associationID, unsigned short streamID,
                   unsigned char *buffer, unsigned int length, unsigned int protocolId,
                   short destAddressIndex, unsigned int context,
                   unsigned int lifetime, int unorderedDelivery,
                   int dontBundle)
```

5.4.6 sctp_setPrimary()

sctp_setPrimary() changes the primary path of an association. The function takes the following parameters:

`associationID` ID of assocation.

`destAddressIndex` index to the new primary path

The function returns an error code: 0 on success, 1 on error (i.e. no assoc, path with this index does not exist etc.).
Definition:

```
short sctp_setPrimary(unsigned int associationID, short destAddressIndex)
```

5.4.7 sctp_receive()

sctp_receive() is called in response to the `Data Arrive`-Notification to get the received data. The function takes the following parameters:

`associationID` ID of association.

`streamID` the stream on which the data chunk is received.

`buffer` pointer to where payload data of arrived chunk will be copied

`length` length of chunk data.

It returns 1 if association does not exist, 0 if okay.
Definition:

```
unsigned short sctp_receive(unsigned int associationID,
            unsigned short streamID, unsigned char *buffer, unsigned int *length)
```

5.4.8 sctp_status()

sctp_status() returns the status of an association (a big struct with a lot of data). This function also allocates some memory, that the application will need to free after having processed all of that datar !! So BEWARE ! See also section 5.2.4. The function takes the following parameters:

`associationID` ID of the assocation.

The function returns a pointer to a newly allocated association **struct AssociationStatus** or NULL if association does not exist.
Definition:

```
AssociationStatus *sctp_status(unsigned int associationID)
```

5.4.9 sctp_changeHeartBeat()

sctp_changeHeartBeat() turns the heartbeat of an association on or off, and sets the interval, if it is turned on. The implementation is currently missing the jitter, that is required by the RFC [1]. The function takes the following parameters:

`associationID` ID of assocation.

`destAddressIndex` index of the address for which HB is turned on/off.

`heartbeatON` turn heartbeat on or off. Use constants `HEARTBEAT_ON` or `HEARTBEAT_OFF`.

`timeIntervall` heartbeat time intervall in milliseconds

The function returns an error code, 0 for success, 1 if association does not exist, or destination address does not exist.

Definition:

```
int sctp_changeHeartBeat(unsigned int associationID,
                    short destAddressIndex, int heartbeatON,
                    unsigned int timeIntervall)
```

5.4.10 sctp_requestHeartbeat()

sctp_requestHeartbeat() sends a heartbeat to the given address of an association. This is like an explicit request from the ULP to the SCTP instance to perform an on-demand heartbeat. The function takes the following parameters:

`associationID` ID of assocation.

`destAddressIndex` destination address to which the heartbeat shall be sent.

Function returns error code (0 == success, 1 == error).

Definition:

```
int sctp_requestHeartbeat(unsigned int associationID, short destAddressIndex);
```

5.4.11 sctp_getSRTT_Report()

The function **sctp_requestHeartbeat**() returns the smoothed round trip time value in milliseconds for a given adress in a given association.

`associationID` ID of assocation.

`destAddressIndex` destination address for which to return the smoothed RTT

Function returns error code (0 == success, 1 == error).

Definition:

```
\bv
unsigned int sctp_getSRTT_Report(unsigned int associationID,
                                 short destAddressIndex);
```

5.4.12 sctp_setFailureThreshold()

sctp_setFailureThreshold is currently NOT implemented ! It will be used to set the threshold for retransmissions on the given address of an association. If the threshold is exceded, the the destination address is considered unreachable. The function takes the following parameters:

`associationID` ID of assocation.

`destAddressIndex` destination address to get new threshold.

`pathMaxRetransmissions` new threshold for retransmissions.

Function currently not implemented, so it returns -1, always.

Definition:

```
int sctp_setFailureThreshold(unsigned int associationID,
                             short destAddressIndex,
                             unsigned short pathMaxRetransmissions)
```

5.4.13 sctp_getInstanceParams()

This function may be used to read the current set of values for parameters that are SCTP instance specific. These may then be changed, and passed to a subsequent call to **sctp_setInstanceParams**().

`SCTP_InstanceName` name of the SCTP instance

The function returns a pointer to an **InstanceParams**-structure. That structure must be freed by the application after this function call (i.e. a malloc() is used by the `libsctp`-library).
Definition:

```
InstanceParams* sctp_getInstanceParams(unsigned short SCTP_InstanceName);
```

5.4.14 sctp_setInstanceParams()

This function is used to set parameters per SCTP instance. As to the parameters that can be set see section 5.2.2.

`SCTP_InstanceName` name of the SCTP instance

`params` pointer to a **InstanceParams**-structure containing new parameters

Will return an error code, -1 if instance does not exist, 0 for success. Function currently not implemented, so it returns -1, always.
Definition:

```
int sctp_setInstanceParams(unsigned short SCTP_InstanceName,
                           InstanceParams* params);
```

5.4.15 sctp_getAssociationParams()

This function may be used to read the current set of values for parameters that are association specific. These may then be changed, and passed to a subsequent call to **sctp_setAssociationParams**().

`associationID` ID of assocation.

`path_id` destination address index from which to read parameters

The function returns a pointer to an **AssocParams**-structure. That structure must be freed by the application after this function call (i.e. a malloc() is used by the `libsctp`-library).
Definition:

```
AssocParams* sctp_getAssociationParams(unsigned int associationID,
                                       unsigned short path_id);
```

5.4.16 sctp_setAssociationParams()

sctp_setAssociationParams() is currently NOT implemented ! Will set protocol parameters of an association. Most of the parameters are path-specific, so you may specify the path for which to set these. The parameters per association (not per path of an association) are set every time you call this function. The function takes the following parameters:

`associationID` ID of assocation.

`path_id` destination address that gets its params set

`params` pointer to an **AssocParams**-structure containing the new parameters

Function currently not implemented, so it returns -1, always. Will return an error code, -1 if instance does not exist, 0 for success.
Definition:

```
int sctp_setAssociationParams(unsigned int associationID,
                              unsigned short path_id
                              AssocParams * params);
```

5.4.17 sctp_receive_unsent()

sctp_receive_unsent() is currently NOT implemented ! It will return messages that could not be sent for some reason, e.g. because association was aborted before they could be transmitted for the first time. The function takes the following parameters:

`associationID` ID of assocation.

`buffer` pointer to a buffer that the application needs to pass. Data is copied there.

`length` size of the buffer passed by application.

`streamID` pointer to the stream id, where data should have been sent.

`streamSN` pointer to stream sequence number of the data chunk that was not sent.

`protocolID` pointer to the protocol ID of the unsent chunk

Function currently not implemented, so it returns -1, always.
Definition:

```
int sctp_receive_unsent(unsigned int associationID, unsigned char *buffer,
                        unsigned int length, unsigned short *streamID,
                        unsigned short *streamSN,unsigned int* protocolId)
```

5.4.18 sctp_receive_unacked()

sctp_receive_unacked() is currently NOT implemented ! Will return messages that were sent, but have not been acknowlegded by the peer before the association was terminated (or aborted). The function takes the following parameters:

`associationID` ID of assocation.

`buffer` pointer to a buffer that the application needs to pass. Data is copied there.

`length` size of the buffer passed by application.

`streamID` pointer to the stream id, where data should have been sent.

`streamSN` pointer to stream sequence number of the data chunk that was not acked

`protocolID` pointer to the protocol ID of the unacked chunk

Function currently not implemented, so it returns -1, always.
Definition:

```
int sctp_receive_unacked(unsigned int associationID, unsigned char *buffer,
                        unsigned int length, unsigned short *streamID,
                        unsigned short *streamSN,unsigned int* protocolId)
```

5.4.19 sctp_unregisterSCTP_instance()

This function removes a registered SCTP instance from the list of instances. If no instances are registered for either IPv4 or IPv6 raw sockets, callbacks for these sockets are also removed.
There are a few points that may need to be checked here:

- Check that all resources belonging to an instance are correctly released. What about chunks that are still in the queues ?
- when the last instance is de-registered, the application should terminate !?

The function only takes one parameter:

`instance_name` ID of the SCTP instance, that was returned when calling **sctp_registerSCTP_instance**().

The function returns an error code, 0 for successful removal, -1 for error (i.e. no such instance).
Definition:

```
int sctp_unregisterSCTP_instance(unsigned short instance_name)
```

5.5 SCTP-to-ULP

All notifcations are realized via callbacks, i.e. upon registering a new SCTP instance, the ULP has to pass a struct containing pointers to a set of functions, that will be called when the respective event occurs, with appropriate parameters.

The `Communication Up` notification may return a pointer to a data structure that the upper layer is interested in. This pointer is subsequently passed along transparently with all callbacks.

5.5.1 Data Arrive Notification

Indicates that new data has arrived from peer (chapter 10.2.A). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned int stream ID

unsigned int length of data

unsigned int protocol ID

unsigned int unordered flag (TRUE==1==unordered, FALSE==0==normal, numbered chunk)

void* pointer to upper layer data

Definition:

```
void (*dataArriveNotif) (unsigned int, unsigned int, unsigned int,
                         unsigned int, unsigned int, void*);
```

5.5.2 Send Failure Indication

Indicates a send failure, i.e. the data could not be sent for some reason (chapter 10.2.B). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned char * pointer to data that was not sent

unsigned int length of data

unsigned int * pointer to context from sctp_send

void* pointer to upper layer data

Definition:

```
void (*sentFailureNotif) (unsigned int, unsigned char *, unsigned int, unsigned int *, void*);
```

5.5.3 Network Status Change Indication

Indicates a change of network status, i.e. a path has changed from state ACTIVE to INACTIVE, or vice versa. (chapter 10.2.C). If there is only *one* active path left, which turns INACTIVE, a COMMUNICATION_ LOST-notification is given instead of a NETWORK STATUS CHANGE. The parameters passed with this callback are (in this order):

unsigned int association ID

short index of the destination address concerned

unsigned short new path state (0==PM_ACTIVE, 1==PM_INACTIVE)

void* pointer to upper layer data

Definition:

```
void (*networkStatusChangeNotif) (unsigned int, short, unsigned short, void*);
```

5.5.4 Communication-Up Notification

Indicates that an association has been successfully established (chapter 10.2.D). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned short status, type of event; the following events are defined:

- COMM_UP_RECEIVED_VALID_COOKIE == 1
- COMM_UP_RECEIVED_COOKIE_ACK == 2
- COMM_UP_RECEIVED_COOKIE_RESTART == 3

union sockunion * pointer to array of destination addresses

int number of destination addresses

unsigned short number input streamns

unsigned short number output streams

void* pointer to upper layer data

The function may return a pointer that the upper layer is intersted in. This pointer will then be passed along with all subsequent notification calls.
Definition:

```
void* (*communicationUpNotif) (unsigned int, unsigned short,
                               union sockunion *, int,
                               unsigned short, unsigned short, void*);
```

5.5.5 Communication-Lost Notification

Indicates that the association with the peer was closed for some reason (chapter 10.2.E). The parameters passed with this callback are (in this order):

unsigned int association ID

unsigned short status, type of event; the following events are defined:

- COMM_LOST_NORMAL == 0
- COMM_LOST_ABORTED == 1
- COMM_LOST_ENDPOINT_UNREACHABLE == 2
- COMM_LOST_EXCEEDED_RETRANSMISSIONS == 3
- COMM_LOST_NO_TCB == 4
- COMM_LOST_ZERO_STREAMS == 8
- COMM_LOST_FAILURE == 9

FIXME: Describe exact semantics, when these are called !

void* pointer to upper layer data

Definition:

```
void (*communicationLostNotif) (unsigned int, unsigned short, void*);
```

5.5.6 Communication Error Notification

Indicates that communication had an error (chapter 10.2.F). Currently not implemented !

unsigned int association ID

unsigned short status, type of error

void* pointer to upper layer data

Definition:

```
void (*communicationErrorNotif) (unsigned int, unsigned short, void*);
```

5.5.7 RestartIndication

Indicates that a RESTART has occurred (chapter 10.2.G). The notification has been added to our implementation, lately but the restart scenario is still somewhat untested. So we need feedback here !

unsigned int association ID

void* pointer to upper layer data

Definition:

```
void (*RestartNotif) (unsigned int, void*);
```

5.5.8 Shutdown Complete Indication

Indicates that a Shutdown has been successfully completed (chapter 10.2.H).

unsigned int association ID

void* pointer to upper layer data

Definition:

```
void (*ShutdownCompleteNotif) (unsigned int, void*);
```

6 Example Programs

6.1 A Discard Server

To be filled in.

6.2 An Echo Server

To be filled in.

References

[1] Stewart, R.R. and Xie, Q. and others: **RFC 2960 - Stream Control Transmission Protocol**, IETF, Network Working Group, October 2000