Network Working Group
Internet-Draft
Expires: August 25, 2005

R. Stewart
Cisco Systems, Inc.
Q. Xie
Motorola, Inc.
L. Yarroll
TimeSys Corp
J. Wood
DoCoMo USA Labs
K. Poon
Sun Microsystems, Inc.
M. Tuexen
Univ. of Applied Sciences Muenster
February 21, 2005

Sockets API Extensions for Stream Control Transmission Protocol (SCTP)

draft-ietf-tsvwg-sctpsocket-10.txt

Status of this Memo

Copyright Notice

Abstract

This document describes a mapping of the Stream Control Transmission Protocol SCTP RFC2960 [8] into a sockets API. The benefits of this mapping include compatibility for TCP applications, access to new SCTP features and a consolidated error and event notification scheme.

Table of Contents

1. Introduction

The sockets API has provided a standard mapping of the Internet Protocol suite to many operating systems. Both TCP RFC793 [1] and UDP RFC768 [2] have benefited from this standard representation and access method across many diverse platforms. SCTP is a new protocol that provides many of the characteristics of TCP but also incorporates semantics more akin to UDP. This document defines a method to map the existing sockets API for use with SCTP, providing both a base for access to new features and compatibility so that most existing TCP applications can be migrated to SCTP with few (if any) changes.

There are three basic design objectives:

1) Maintain consistency with existing sockets APIs:
We define a sockets mapping for SCTP that is consistent with other sockets API protocol mappings (for instance, UDP, TCP, IPv4, and IPv6).
2) Support a one-to-many style interface
This set of semantics is similar to that defined for connection-less protocols, such as UDP. A one-to-many style SCTP socket should be able to control multiple SCTP associations. This is similar to an UDP socket, which can communicate with many peer end points. Each of these associations is assigned an association ID so that an applications can use the ID to differentiate them. Note that SCTP is connection-oriented in nature, and it does not support broadcast or multicast communications, as UDP does.
3) Support a one-to-one style interface
This interface supports a similar semantics as sockets for connection-oriented protocols, such as TCP. A one-to-one style SCTP socket should only control one SCTP association.
One purpose of defining this interface is to allow existing applications built on other connection-oriented protocols be ported to use SCTP with very little effort. And developers familiar with those semantics can easily adapt to SCTP. Another purpose is to make sure that existing mechanisms in most OSes to deal with socket, such as select(), should continue to work with this style of socket.
Extensions are added to this mapping to provide mechanisms to exploit new features of SCTP.

Goals 2 and 3 are not compatible, so in this document we define two modes of mapping, namely the one-to-many style mapping and the one-to-one style mapping. These two modes share some common data structures and operations, but will require the use of two different application programming styles. Note that all new SCTP features can be used with both styles of socket. The decision on which one to use

depends mainly on the nature of applications.

A mechanism is defined to extract a one-to-many style SCTP association into a one-to-one style socket.

Some of the SCTP mechanisms cannot be adequately mapped to existing socket interface. In some cases, it is more desirable to have new interface instead of using existing socket calls. Section 8 of this document describes those new interface.

2. Conventions

2.1 Data Types

Whenever possible, data types from Draft 6.6 (March 1997) of POSIX 1003.1g are used: uintN_t means an unsigned integer of exactly N bits (e.g., uint16_t). We also assume the argument data types from 1003.1g when possible (e.g., the final argument to setsockopt() is a size_t value). Whenever buffer sizes are specified, the POSIX 1003.1 size_t data type is used.

3. one-to-many style Interface

The one-to-many style interface has the following characteristics:

A) Outbound association setup is implicit.

B) Messages are delivered in complete messages (with one notable exception).

C) There is a 1 to MANY relationship between socket and association.

3.1 Basic Operation

A typical server in this style uses the following socket calls in sequence to prepare an endpoint for servicing requests:

1. socket()
2. bind()
3. listen()
4. recvmsg()
5. sendmsg()
6. close()

A typical client uses the following calls in sequence to setup an association with a server to request services:

1. socket()
2. sendmsg()
3. recvmsg()
4. close()

In this style, by default, all the associations connected to the endpoint are represented with a single socket. Each associations is assigned an association ID (type is sctp_assoc_t) so that an application can use it to differentiate between them. In some implementations, the peer end point's addresses can also be used for this purpose. But this is not required for performance reasons. If an implementation does not support using addresses to differentiate between different associations, the sendto() call can only be used to setup an association implicitly. It cannot be used to send data to an established association as the association ID cannot be specified.

Once as association ID is assigned to an SCTP association, that ID will not be reused until the application explicitly terminates the association. The resources belonged to that association will not be freed until that happens. This is similar to the close() operation on a normal socket. The only exception is when the SCTP_AUTOCLOSE option (section 7.1.8) is set. In this case, after the association

is terminated automatically, the association ID assigned to it can be reused. All applications using this option should be aware of this to avoid the possible problem of sending data to an incorrect peer end point.

If the server or client wishes to branch an existing association off to a separate socket, it is required to call sctp_peeloff() and in the parameter specifies the association identification. The sctp_peeloff() call will return a new socket which can then be used with recv() and send() functions for message passing. See Section 8.2 for more on branched-off associations.

Once an association is branched off to a separate socket, it becomes completely separated from the original socket. All subsequent control and data operations to that association must be done through the new socket. For example, the close operation on the original socket will not terminate any associations that have been branched off to a different socket.

We will discuss the one-to-many style socket calls in more details in the following subsections.

3.1.1 socket() - one-to-many style socket

Applications use socket() to create a socket descriptor to represent an SCTP endpoint.

The syntax is,

```
sd = socket(PF_INET, SOCK_SEQPACKET, IPPROTO_SCTP);
```

or,

```
sd = socket(PF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP);
```

Here, SOCK_SEQPACKET indicates the creation of a one-to-many style socket.

The first form creates an endpoint which can use only IPv4 addresses, while, the second form creates an endpoint which can use both IPv6 and IPv4 addresses.

3.1.2 bind() - one-to-many style socket

Applications use bind() to specify which local address the SCTP endpoint should associate itself with.

An SCTP endpoint can be associated with multiple addresses. To do

this, sctp_bindx() is introduced in section Section 8.1 to help applications do the job of associating multiple addresses.

These addresses associated with a socket are the eligible transport addresses for the endpoint to send and receive data. The endpoint will also present these addresses to its peers during the association initialization process, see RFC2960 [8].

After calling bind(), if the endpoint wishes to accept new associations on the socket, it must call listen() (see section Section 3.1.3).

The syntax of bind() is,

```
ret = bind(int sd, struct sockaddr *addr, socklen_t addrlen);
```

sd: the socket descriptor returned by socket().
addr: the address structure (struct sockaddr_in or struct
   sockaddr_in6 RFC2553 [7]).
addrlen: the size of the address structure.

If sd is an IPv4 socket, the address passed must be an IPv4 address. If the sd is an IPv6 socket, the address passed can either be an IPv4 or an IPv6 address.

Applications cannot call bind() multiple times to associate multiple addresses to an endpoint. After the first call to bind(), all subsequent calls will return an error.

If addr is specified as a wildcard (INADDR_ANY for an IPv4 address, or as IN6ADDR_ANY_INIT or in6addr_any for an IPv6 address), the operating system will associate the endpoint with an optimal address set of the available interfaces.

If a bind() is not called prior to a sendmsg() call that initiates a new association, the system picks an ephemeral port and will choose an address set equivalent to binding with a wildcard address. One of those addresses will be the primary address for the association. This automatically enables the multi-homing capability of SCTP.

3.1.3 listen() - One-to-many style socket

By default, new associations are not accepted for one-to-many style sockets. An application uses listen() to mark a socket as being able to accept new associations. The syntax is,

```
int listen(int sd, int backlog);
 sd      - the socket descriptor of the endpoint.
```

backlog - if backlog is non-zero, enable listening else disable listening.

Note that one-to-many style socket consumers do not need to call accept to retrieve new associations. Calling accept() on a one-to-many style socket should return EOPNOTSUPP. Rather, new associations are accepted automatically, and notifications of the new associations are delivered via recvmsg() with the SCTP_ASSOC_CHANGE event (if these notifications are enabled). Clients will typically not call listen(), so that they can be assured that the only associations on the socket will be ones they actively initiated. Server or peer-to-peer sockets, on the other hand, will always accept new associations, so a well-written application using server one-to-many style sockets must be prepared to handle new associations from unwanted peers.

Also note that the SCTP_ASSOC_CHANGE event provides the association ID for a new association, so if applications wish to use the association ID as input to other socket calls, they should ensure that the SCTP_ASSOC_CHANGE event is enabled (it is enabled by default).

3.1.4 sendmsg() and recvmsg() - one-to-many style socket

An application uses sendmsg() and recvmsg() call to transmit data to and receive data from its peer.

```
ssize_t sendmsg(int sd, const struct msghdr *message, int flags);

ssize_t recvmsg(int sd, struct msghdr *message, int flags);
```

sd: the socket descriptor of the endpoint.
message: pointer to the msghdr structure which contains a single user message and possibly some ancillary data. See Section 5 for complete description of the data structures.
flags: No new flags are defined for SCTP at this level. See Section 5 for SCTP-specific flags used in the msghdr structure.

As we will see in Section 5, along with the user data, the ancillary data field is used to carry the sctp_sndrcvinfo and/or the sctp_initmsg structures to perform various SCTP functions including specifying options for sending each user message. Those options, depending on whether sending or receiving, include stream number, stream sequence number, various flags, context and payload protocol Id, etc.

When sending user data with sendmsg(), the msg_name field in msghdr structure will be filled with one of the transport addresses of the

intended receiver. If there is no association existing between the sender and the intended receiver, the sender's SCTP stack will set up a new association and then send the user data (see Section 3.2 for more on implicit association setup).

If a peer sends a SHUTDOWN, a SCTP_SHUTDOWN_EVENT notification will be delivered if that notification has been enabled, and no more data can be sent to that association. Any attempt to send more data will cause sendmsg() to return with an ESHUTDOWN error. Note that the socket is still open for reading at this point so it is possible to retrieve notifications.

When receiving a user message with recvmsg(), the msg_name field in msghdr structure will be populated with the source transport address of the user data. The caller of recvmsg() can use this address information to determine to which association the received user message belongs. Note that if SCTP_ASSOC_CHANGE events are disabled, applications must use the peer transport address provided in the msg_name field by recvmsg() to perform correlation to an association, since they will not have the association ID.

If all data in a single message has been delivered, MSG_EOR will be set in the msg_flags field of the msghdr structure (see section Section 5.1).

If the application does not provide enough buffer space to completely receive a data message, MSG_EOR will not be set in msg_flags. Successive reads will consume more of the same message until the entire message has been delivered, and MSG_EOR will be set.

If the SCTP stack is running low on buffers, it may partially deliver a message. In this case, MSG_EOR will not be set, and more calls to recvmsg() will be necessary to completely consume the message. Only one message at a time can be partially delivered.

Note, if the socket is a branched-off socket that only represents one association (see Section 3.1), the msg_name field can be used to override the primary address when sending data.

3.1.5 close() - one-to-many style socket

Applications use close() to perform graceful shutdown (as described in Section 10.1 of RFC2960 [8]) on ALL the associations currently represented by a one-to-many style socket.

The syntax is:

```
ret = close(int sd);
```

sd - the socket descriptor of the associations to be closed.

To gracefully shutdown a specific association represented by the one-to-many style socket, an application should use the sendmsg() call, and including the MSG_EOF flag. A user may optionally terminate an association non-gracefully by sending with the MSG_ABORT flag and possibly passing a user specified abort code in the data field. Both flags MSG_EOF and MSG_ABORT are passwd with ancillary data (see Section 5.2.2) in the sendmsg call.

If sd in the close() call is a branched-off socket representing only one association, the shutdown is performed on that association only.

3.1.6 connect() - one-to-many style socket

An application may use the connect() call in the one-to-many style to initiate an association without sending data.

The syntax is:

```
ret = connect(int sd, const struct sockaddr *nam, socklen_t len);
```

sd: the socket descriptor to have a new association added to.
nam: the address structure (either struct sockaddr_in or struct sockaddr_in6 defined in RFC2553 [7]).
len: the size of the address.
Multiple connect() calls can be made on the same socket to create multiple associations. This is different from the semantics of connect() on a UDP socket.

3.2 Implicit Association Setup

Once the bind() call is complete on a one-to-many style socket, the application can begin sending and receiving data using the sendmsg()/recvmsg() or sendto()/recvfrom() calls, without going through any explicit association setup procedures (i.e., no connect() calls required).

Whenever sendmsg() or sendto() is called and the SCTP stack at the sender finds that there is no association existing between the sender and the intended receiver (identified by the address passed either in the msg_name field of msghdr structure in the sendmsg() call or the dest_addr field in the sendto() call), the SCTP stack will automatically setup an association to the intended receiver.

Upon the successful association setup a SCTP_COMM_UP notification will be dispatched to the socket at both the sender and receiver side. This notification can be read by the recvmsg() system call

(see Section 3.1.3).

Note, if the SCTP stack at the sender side supports bundling, the first user message may be bundled with the COOKIE ECHO message RFC2960 [8].

When the SCTP stack sets up a new association implicitly, it first consults the sctp_initmsg structure, which is passed along within the ancillary data in the sendmsg() call (see Section 5.2.1 for details of the data structures), for any special options to be used on the new association.

If this information is not present in the sendmsg() call, or if the implicit association setup is triggered by a sendto() call, the default association initialization parameters will be used. These default association parameters may be set with respective setsockopt() calls or be left to the system defaults.

Implicit association setup cannot be initiated by send()/recv() calls.

3.3 Non-blocking mode

Some SCTP users might want to avoid blocking when they call socket interface function.

Once all bind() calls are complete on a one-to-many style socket, the application must set the non-blocking option by a fcntl() (such as O_NONBLOCK). After which the sendmsg() function returns immediately, and the success or failure of the data message (and possible SCTP_INITMSG parameters) will be signaled by the SCTP_ASSOC_CHANGE event with SCTP_COMM_UP or CANT_START_ASSOC. If user data could not be sent (due to a CANT_START_ASSOC), the sender will also receive a SCTP_SEND_FAILED event. Those event(s) can be received by the user calling of recvmsg(). A server (having called listen()) is also notified of an association up event by the reception of a SCTP_ASSOC_CHANGE with SCTP_COMM_UP via the calling of recvmsg() and possibly the reception of the first data message.

In order to shutdown the association gracefully, the user must call sendmsg() with no data and with the MSG_EOF flag set. The function returns immediately, and completion of the graceful shutdown is indicated by an SCTP_ASSOC_CHANGE notification of type SHUTDOWN_COMPLETE (see Section 5.3.1.1). Note that this can also be done using the sctp_send() call described in Section 8.10.

An application is recommended to use caution when using select() (or poll()) for writing on a one-to-many style socket. The reason being

that interpretation of select on write is implementation specific. Generally a positive return on a select on write would only indicate that one of the associations represented by the one-to-many socket is writable. An application that writes after the select return may still block since the association that was writeable is not the destination association of the write call. Likewise select (or poll()) for reading from a one-to-many socket will only return an indication that one of the associations represented by the socket has data to be read.

An application that wishes to know that a particular association is ready for reading or writing should either use the one-to-one style or use the sctp_peelloff() (see Section 8.2) function to seperate the association of interest from the one-to-many socket.

3.4 Special considerations

The fact that a one-to-many style socket can provide access to many SCTP associations through a single socket descriptor has important implications for both application programmers and system programmers implementing this API. A key issue is how buffer space inside the sockets layer is managed. Because this implementation detail directly affects how application programmers must write their code to ensure correct operation and portability, this section provides some guidance to both implementors and application programmers.

An important feature that SCTP shares with TCP is flow control: specifically, a sender may not send data faster than the receiver can consume it.

For TCP, flow control is typically provided for in the sockets API as follows. If the reader stops reading, the sender queues messages in the socket layer until it uses all of its socket buffer space allocation creating a "stalled connection". Further attempts to write to the socket will block or return the error EAGAIN or EWOULDBLOCK for a non-blocking socket. At some point, either the connection is closed, or the receiver begins to read again freeing space in the output queue.

For one-to-one style SCTP sockets (this includes sockets descriptors that were separated from a one-to-many style socket with sctp_peeloff()) the behavior is identical. For one-to-many style SCTP sockets, the fact that we have multiple associations on a single socket makes the situation more complicated. If the implementation uses a single buffer space allocation shared by all associations, a single stalled association can prevent the further sending of data on all associations active on a particular one-to-many style socket.

For a blocking socket, it should be clear that a single stalled association can block the entire socket. For this reason, application programmers may want to use non-blocking one-to-many style sockets. The application should at least be able to send messages to the non-stalled associations.

But a non-blocking socket is not sufficient if the API implementor has chosen a single shared buffer allocation for the socket. A single stalled association would eventually cause the shared allocation to fill, and it would become impossible to send even to non-stalled associations.

The API implementor can solve this problem by providing each association with its own allocation of outbound buffer space. Each association should conceptually have as much buffer space as it would have if it had its own socket. As a bonus, this simplifies the implementation of sctp_peeloff().

To ensure that a given stalled association will not prevent other non-stalled associations from being writable, application programmers should either:

(a) demand that the underlying implementation dedicates independent buffer space allotments to each association (as suggested above), or

(b) verify that their application layer protocol does not permit large amounts of unread data at the receiver (this is true of some request-response protocols, for example), or

(c) use one-to-one style sockets for association which may potentially stall (either from the beginning, or by using sctp_peeloff before sending large amounts of data that may cause a stalled condition).

An implemenation which dedicates independent buffer space for each association should define HAVE_SCTP_MULTIBUF to 1.

4. one-to-one style Interface

The goal of this style is to follow as closely as possible the current practice of using the sockets interface for a connection oriented protocol, such as TCP. This style enables existing applications using connection oriented protocols to be ported to SCTP with very little effort.

Note that some new SCTP features and some new SCTP socket options can only be utilized through the use of sendmsg() and recvmsg() calls, see Section 4.1.8. Also note that some socket interfaces may not be able to provide data on the third leg of the association set up with this interface style.

4.1 Basic Operation

A typical server in one-to-one style uses the following system call sequence to prepare an SCTP endpoint for servicing requests:

1. socket()

2. bind()

3. listen()

4. accept()

The accept() call blocks until a new association is set up. It returns with a new socket descriptor. The server then uses the new socket descriptor to communicate with the client, using recv() and send() calls to get requests and send back responses.

Then it calls

5. close()

to terminate the association.

A typical client uses the following system call sequence to setup an association with a server to request services:

1. socket()

2. connect()

After returning from connect(), the client uses send() and recv() calls to send out requests and receive responses from the server.

The client calls

3. close()

to terminate this association when done.

4.1.1 socket() - one-to-one style socket

Applications calls socket() to create a socket descriptor to represent an SCTP endpoint.

The syntax is:

```
int socket(PF_INET, SOCK_STREAM, IPPROTO_SCTP);

 or,

int socket(PF_INET6, SOCK_STREAM, IPPROTO_SCTP);
```

Here, SOCK_STREAM indicates the creation of a one-to-one style socket.

The first form creates an endpoint which can use only IPv4 addresses, while the second form creates an endpoint which can use both IPv6 and IPv4 addresses.

4.1.2 bind() - one-to-one style socket

Applications use bind() to pass an address to be associated with an SCTP endpoint to the system. bind() allows only either a single address or a IPv4 or IPv6 wildcard address to be bound. An SCTP endpoint can be associated with multiple addresses. To do this, sctp_bindx() is introduced in Section 8.1 to help applications do the job of associating multiple addresses.

These addresses associated with a socket are the eligible transport addresses for the endpoint to send and receive data. The endpoint will also present these addresses to its peers during the association initialization process, see RFC2960 [8].

The syntax is:

```
int bind(int sd, struct sockaddr *addr, socklen_t addrlen);
```

sd: the socket descriptor returned by socket() call.

addr: the address structure (either struct sockaddr_in or struct sockaddr_in6 defined in RFC2553 [7]).
addrlen: the size of the address structure.

If sd is an IPv4 socket, the address passed must be an IPv4 address. Otherwise, i.e., the sd is an IPv6 socket, the address passed can either be an IPv4 or an IPv6 address.

Applications cannot call bind() multiple times to associate multiple addresses to the endpoint. After the first call to bind(), all subsequent calls will return an error.

If addr is specified as a wildcard (INADDR_ANY for an IPv4 address, or as IN6ADDR_ANY_INIT or in6addr_any for an IPv6 address), the operating system will associate the endpoint with an optimal address set of the available interfaces.

If a bind() is not called prior to the connect() call, the system picks an ephemeral port and will choose an address set equivalent to binding with a wildcard address. One of those addresses will be the primary address for the association. This automatically enables the multi-homing capability of SCTP.

The completion of this bind() process does not ready the SCTP endpoint to accept inbound SCTP association requests. Until a listen() system call, described below, is performed on the socket, the SCTP endpoint will promptly reject an inbound SCTP INIT request with an SCTP ABORT.

4.1.3 listen() - one-to-one style socket

Applications use listen() to ready the SCTP endpoint for accepting inbound associations.

The syntax is:

```
int listen(int sd, int backlog);
```

sd: the socket descriptor of the SCTP endpoint.
backlog: this specifies the max number of outstanding associations allowed in the socket's accept queue. These are the associations that have finished the four-way initiation handshake (see Section 5 of RFC2960 [8]) and are in the ESTABLISHED state. Note, a backlog of '0' indicates that the caller no longer wishes to receive new associations.

4.1.4 accept() - one-to-one style socket

Applications use accept() call to remove an established SCTP association from the accept queue of the endpoint. A new socket descriptor will be returned from accept() to represent the newly formed association.

The syntax is:

```
new_sd = accept(int sd, struct sockaddr *addr, socklen_t *addrlen);
```

new_sd: the socket descriptor for the newly formed association.
sd the listening socket descriptor.
addr on return, will contain the primary address of the peer endpoint.
addrlen on return, will contain the size of addr.

4.1.5 connect() - one-to-one style socket

Applications use connect() to initiate an association to a peer.

The syntax is:

```
int connect(int sd, const struct sockaddr *addr, socklen_t addrlen);
```

sd: the socket descriptor of the endpoint.
addr the peer's address.
addrlen the size of the address.

This operation corresponds to the ASSOCIATE primitive described in section 10.1 of RFC2960 [8].

By default, the new association created has only one outbound stream. The SCTP_INITMSG option described in Section 7.1.3 should be used before connecting to change the number of outbound streams.

If a bind() is not called prior to the connect() call, the system picks an ephemeral port and will choose an address set equivalent to binding with INADDR_ANY and IN6ADDR_ANY for IPv4 and IPv6 socket respectively. One of those addresses will be the primary address for the association. This automatically enables the multi-homing capability of SCTP.

Note that SCTP allows data exchange, similar to T/TCP RFC1644 [3], during the association set up phase. If an application wants to do this, it cannot use connect() call. Instead, it should use sendto() or sendmsg() to initiate an association. If it uses sendto() and it wants to change initialization behavior, it needs to use the

SCTP_INITMSG socket option before calling sendto(). Or it can use SCTP_INIT type sendmsg() to initiate an association without doing the setsockopt(). Note that some sockets implementations may not support the sending of data to initiate an assocation with the one-to-one style (implementations that do not support T/TCP normally have this restriction). Implementations which allow sending of data to initiate an association without calling connect() define the preprocessor constant HAVE_SCTP_NOCONNECT to 1.

SCTP does not support half close semantics. This means that unlike T/TCP, MSG_EOF should not be set in the flags parameter when calling sendto() or sendmsg() when the call is used to initiate a connection. MSG_EOF is not an acceptable flag with SCTP socket.

4.1.6 close() - one-to-one style socket

Applications use close() to gracefully close down an association.

The syntax is:

```
int close(int sd);

   sd      - the socket descriptor of the association to be closed.
```

After an application calls close() on a socket descriptor, no further socket operations will succeed on that descriptor.

4.1.7 shutdown() - one-to-one style socket

SCTP differs from TCP in that it does not have half closed semantics. Hence the shutdown() call for SCTP is an approximation of the TCP shutdown() call, and solves some different problems. Full TCP-compatibility is not provided, so developers porting TCP applications to SCTP may need to recode sections that use shutdown(). (Note that it is possible to achieve the same results as half close in SCTP using SCTP streams.)

The syntax is:

```
int shutdown(int sd, int how);

 sd      - the socket descriptor of the association to be closed.

 how     - Specifies the type of shutdown.  The  values  are
           as follows:

           SHUT_RD
                 Disables further receive operations. No SCTP
                 protocol action is taken.

           SHUT_WR
                 Disables further send operations, and initiates
                 the SCTP shutdown sequence.

           SHUT_RDWR
                 Disables further send  and  receive  operations
                 and initiates the SCTP shutdown sequence.
```

The major difference between SCTP and TCP shutdown() is that SCTP SHUT_WR initiates immediate and full protocol shutdown, whereas TCP SHUT_WR causes TCP to go into the half closed state. SHUT_RD behaves the same for SCTP as TCP. The purpose of SCTP SHUT_WR is to close the SCTP association while still leaving the socket descriptor open, so that the caller can receive back any data SCTP was unable to deliver (see Section 5.3.1.4 for more information).

To perform the ABORT operation described in RFC2960 [8] section 10.1, an application can use the socket option SO_LINGER. It is described in Section 7.1.4.

4.1.8 sendmsg() and recvmsg() - one-to-one style socket

With a one-to-one style socket, the application can also use sendmsg() and recvmsg() to transmit data to and receive data from its peer. The semantics is similar to those used in the one-to-many style (section Section 3.1.3), with the following differences:

1) When sending, the msg_name field in the msghdr is not used to specify the intended receiver, rather it is used to indicate a preferred peer address if the sender wishes to discourage the stack from sending the message to the primary address of the receiver. If the transport address given is not part of the current association, the data will not be sent and a SCTP_SEND_FAILED event will be delivered to the application if send failure events are enabled.

4.1.9 getpeername()

Applications use getpeername() to retrieve the primary socket address of the peer. This call is for TCP compatibility, and is not multi-homed. It does not work with one-to-many style sockets. See Section 8.3 for a multi-homed/one-to-many style version of the call .

The syntax is:

```
int getpeername(int sd, struct sockaddr *address,
                socklen_t *len);
```

```
sd      - the socket descriptor to be queried.

address - On return, the peer primary address is stored in
          this buffer. If the socket is an IPv4 socket, the
          address will be IPv4. If the socket is an IPv6 socket,
          the address will be either an IPv6 or IPv4
          address.

len     - The caller should set the length of address here.
          On return, this is set to the length of the returned
          address.
```

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

5. Data Structures

We discuss in this section important data structures which are specific to SCTP and are used with sendmsg() and recvmsg() calls to control SCTP endpoint operations and to access ancillary information and notifications.

5.1 The msghdr and cmsghdr Structures

The msghdr structure used in the sendmsg() and recvmsg() calls, as well as the ancillary data carried in the structure, is the key for the application to set and get various control information from the SCTP endpoint.

The msghdr and the related cmsghdr structures are defined and discussed in details in RFC2292 [6]. Here we will cite their definitions from RFC2292 [6].

The msghdr structure:

```
struct msghdr {
  void      *msg_name;        /* ptr to socket address structure */
  socklen_t  msg_namelen;     /* size of socket address structure */
  struct iovec  *msg_iov;     /* scatter/gather array */
  size_t     msg_iovlen;      /* # elements in msg_iov */
  void      *msg_control;     /* ancillary data */
  socklen_t  msg_controllen;  /* ancillary data buffer length */
  int        msg_flags;       /* flags on received message */
};
```

The cmsghdr structure:

```
struct cmsghdr {
  socklen_t  cmsg_len;   /* #bytes, including this header */
  int        cmsg_level; /* originating protocol */
  int        cmsg_type;  /* protocol-specific type */
             /* followed by unsigned char cmsg_data[]; */
  };
```

In the msghdr structure, the usage of msg_name has been discussed in previous sections (see Section 3.1.3 and Section 4.1.8).

The scatter/gather buffers, or I/O vectors (pointed to by the msg_iov field) are treated as a single SCTP data chunk, rather than multiple chunks, for both sendmsg() and recvmsg().

The msg_flags are not used when sending a message with sendmsg().

If a notification has arrived, recvmsg() will return the notification with the MSG_NOTIFICATION flag set in msg_flags. If the MSG_NOTIFICATION flag is not set, recvmsg() will return data. See Section 5.3 for more information about notifications.

If all portions of a data frame or notification have been read, recvmsg() will return with MSG_EOR set in msg_flags.

5.2 SCTP msg_control Structures

A key element of all SCTP-specific socket extensions is the use of ancillary data to specify and access SCTP-specific data via the struct msghdr's msg_control member used in sendmsg() and recvmsg(). Fine-grained control over initialization and sending parameters are handled with ancillary data.

Each ancillary data item is proceeded by a struct cmsghdr (see Section 5.1), which defines the function and purpose of the data contained in in the cmsg_data[] member.

There are two kinds of ancillary data used by SCTP: initialization data, and, header information (SNDRCV). Initialization data (one-to-many style only) sets protocol parameters for new associations. Section 5.2.1 provides more details. Header information can set or report parameters on individual messages in a stream. See Section 5.2.2 for how to use SNDRCV ancillary data.

By default on a one-to-one style socket, SCTP will pass no ancillary data; on a one-to-many style socket, SCTP will only pass SCTP_SNDRCV and SCTP_ASSOC_CHANGE information. Specific ancillary data items can be enabled with socket options defined for SCTP; see Section 7.3.

Note that all ancillary types are fixed length; see Section 5.4 for further discussion on this. These data structures use struct sockaddr_storage (defined in RFC2553 [7]) as a portable, fixed length address format.

Other protocols may also provide ancillary data to the socket layer consumer. These ancillary data items from other protocols may intermingle with SCTP data. For example, the IPv6 socket API definitions (RFC2292 [6] and RFC2553 [7]) define a number of ancillary data items. If a socket API consumer enables delivery of both SCTP and IPv6 ancillary data, they both may appear in the same msg_control buffer in any order. An application may thus need to handle other types of ancillary data besides that passed by SCTP.

The sockets application must provide a buffer large enough to accommodate all ancillary data provided via recvmsg(). If the buffer

is not large enough, the ancillary data will be truncated and the msghdr's msg_flags will include MSG_CTRUNC.

5.2.1 SCTP Initiation Structure (SCTP_INIT)

This cmsghdr structure provides information for initializing new SCTP associations with sendmsg(). The SCTP_INITMSG socket option uses this same data structure. This structure is not used for recvmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_INIT	struct sctp_initmsg

Here is the definition of the sctp_initmsg structure:

```
struct sctp_initmsg {
   uint16_t sinit_num_ostreams;
   uint16_t sinit_max_instreams;
   uint16_t sinit_max_attempts;
   uint16_t sinit_max_init_timeo;
};
```

sinit_num_ostreams: 16 bits (unsigned integer)

This is an integer number representing the number of streams that the application wishes to be able to send to. This number is confirmed in the SCTP_COMM_UP notification and must be verified since it is a negotiated number with the remote endpoint. The default value of 0 indicates to use the endpoint default value.

sinit_max_instreams: 16 bits (unsigned integer)

This value represents the maximum number of inbound streams the application is prepared to support. This value is bounded by the actual implementation. In other words the user MAY be able to support more streams than the Operating System. In such a case, the Operating System limit overrides the value requested by the user. The default value of 0 indicates to use the endpoint's default value.

sinit_max_attempts: 16 bits (unsigned integer)

This integer specifies how many attempts the SCTP endpoint should make at resending the INIT. This value overrides the system SCTP 'Max.Init.Retransmits' value. The default value of 0 indicates to use the endpoint's default value. This is normally set to the system's default 'Max.Init.Retransmit' value.

sinit_max_init_timeo: 16 bits (unsigned integer)

This value represents the largest Time-Out or RTO value (in milliseconds) to use inattempting a INIT. Normally the 'RTO.Max' is used to limit the doubling of the RTO upon timeout. For the INIT message this value MAY override 'RTO.Max'. This value MUST NOT influence 'RTO.Max' during data transmission and is only used to bound the initial setup time. A default value of 0 indicates to use the endpoint's default value. This is normally set to the system's 'RTO.Max' value (60 seconds).

5.2.2 SCTP Header Information Structure (SCTP_SNDRCV)

This cmsghdr structure specifies SCTP options for sendmsg() and describes SCTP header information about a received message through recvmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_SNDRCV	struct sctp_sndrcvinfo

Here is the definition of sctp_sndrcvinfo:

```
struct sctp_sndrcvinfo {
   uint16_t sinfo_stream;
   uint16_t sinfo_ssn;
   uint16_t sinfo_flags;
   uint32_t sinfo_ppid;
   uint32_t sinfo_context;
   uint32_t sinfo_timetolive;
   uint32_t sinfo_tsn;
   uint32_t sinfo_cumtsn;
   sctp_assoc_t sinfo_assoc_id;
};
```

sinfo_stream: 16 bits (unsigned integer)

For recvmsg() the SCTP stack places the message's stream number in this value. For sendmsg() this value holds the stream number that the application wishes to send this message to. If a sender specifies an invalid stream number an error indication is returned and the call fails.

sinfo_ssn: 16 bits (unsigned integer)

For recvmsg() this value contains the stream sequence number that the remote endpoint placed in the DATA chunk. For fragmented messages this is the same number for all deliveries of the message (if more than one recvmsg() is needed to read the message). The sendmsg()

call will ignore this parameter.

sinfo_ppid: 32 bits (unsigned integer)

This value in sendmsg() is an opaque unsigned value that is passed to the remote end in each user message. In recvmsg() this value is the same information that was passed by the upper layer in the peer application. Please note that byte order issues are NOT accounted for and this information is passed opaquely by the SCTP stack from one end to the other.

sinfo_context: 32 bits (unsigned integer)

This value is an opaque 32 bit context datum that is used in the sendmsg() function. This value is passed back to the upper layer if a error occurs on the send of a message and is retrieved with each undelivered message (Note: if a endpoint has done multiple sends, all of which fail, multiple different sinfo_context values will be returned. One with each user data message).

sinfo_flags: 16 bits (unsigned integer)

This field may contain any of the following flags and is composed of a bitwise OR of these values.

recvmsg() flags:

MSG_UNORDERED - This flag is present when the message was sent non-ordered.

sendmsg() flags:

MSG_UNORDERED - This flag requests the un-ordered delivery of the message. If this flag is clear the datagram is considered an ordered send.

MSG_ADDR_OVER - This flag, in the one-to-many style, requests the SCTP stack to override the primary destination address with the address found with the sendto/sendmsg call.

MSG_ABORT - Setting this flag causes the specified association to abort by sending an ABORT message to the peer (one-to-many style only). The ABORT chunk will contain an error cause 'User Initiated Abort' with cause code 12.The cause specific information of this error cause is provided in msg_iov.

MSG_EOF - Setting this flag invokes the SCTP graceful shutdown procedures on the specified association. Graceful shutdown assures that all data enqueued by both endpoints is successfully transmitted before closing the association (one-to-many style only).

MSG_SENDALL - This flag, if set, will cause a one-to-many model socket to send the message to all associations that are currently established on this socket. For the one-to-one socket, this flag has no effect.

sinfo_timetolive: 32 bit (unsigned integer)

For the sending side, this field contains the message time to live in milliseconds. The sending side will expire the message within the specified time period if the message as not been sent to the peer within this time period. This value will override any default value set using any socket option. Also note that the value of 0 is special in that it indicates no timeout should occur on this message.

sinfo_tsn: 32 bit (unsigned integer)

For the receiving side, this field holds a TSN that was assigned to one of the SCTP Data Chunks.

sinfo_cumtsn: 32 bit (unsigned integer)

This field will hold the current cumulative TSN as known by the underlying SCTP layer. Note this field is ignored when sending and only valid for a receive operation when sinfo_flags are set to MSG_UNORDERED.

sinfo_assoc_id: sizeof (sctp_assoc_t)

The association handle field, sinfo_assoc_id, holds the identifier for the association announced in the SCTP_COMM_UP notification. All notifications for a given association have the same identifier. Ignored for one-to-one style sockets.

A sctp_sndrcvinfo item always corresponds to the data in msg_iov.

5.3 SCTP Events and Notifications

An SCTP application may need to understand and process events and errors that happen on the SCTP stack. These events include network status changes, association startups, remote operational errors and undeliverable messages. All of these can be essential for the application.

When an SCTP application layer does a recvmsg() the message read is normally a data message from a peer endpoint. If the application wishes to have the SCTP stack deliver notifications of non-data events, it sets the appropriate socket option for the notifications it wants. See Section 7.3 for these socket options. When a notification arrives, recvmsg() returns the notification in the application-supplied data buffer via msg_iov, and sets MSG_NOTIFICATION in msg_flags.

This section details the notification structures. Every notification structure carries some common fields which provides general information.

A recvmsg() call will return only one notification at a time. Just as when reading normal data, it may return part of a notification if the msg_iov buffer is not large enough. If a single read is not sufficient, msg_flags will have MSG_EOR clear. The user MUST finish reading the notification before subsequent data can arrive.

5.3.1 SCTP Notification Structure

The notification structure is defined as the union of all notification types.

```
    union sctp_notification {
     struct {
       uint16_t sn_type;             /* Notification type. */
       uint16_t sn_flags;
       uint32_t sn_length;
       } sn_header;
       struct sctp_assoc_change   sn_assoc_change;
       struct sctp_paddr_change   sn_paddr_change;
       struct sctp_remote_error   sn_remote_error;
       struct sctp_send_failed    sn_send_failed;
       struct sctp_shutdown_event sn_shutdown_event;
       struct sctp_adaption_event sn_adaption_event;
       struct sctp_pdapi_event    sn_pdapi_event;
    };
```

sn_type: 16 bits (unsigned integer)

The following list describes the SCTP notification and event types for the field sn_type.

SCTP_ASSOC_CHANGE: This tag indicates that an association has either been opened or closed. Refer to Section 5.3.1.1 for details.

SCTP_PEER_ADDR_CHANGE: This tag indicates that an address that is part of an existing association has experienced a change of state (e.g. a failure or return to service of the reachability of a endpoint via a specific transport address). Please see Section 5.3.1.2 for data structure details.

SCTP_REMOTE_ERROR: The attached error message is an Operational Error received from the remote peer. It includes the complete TLV sent by the remote endpoint. See Section 5.3.1.3 for the detailed format.

SCTP_SEND_FAILED: The attached datagram could not be sent to the remote endpoint. This structure includes the original SCTP_SNDRCVINFO that was used in sending this message i.e. this structure uses the sctp_sndrecvinfo per Section 5.3.1.4.

SCTP_SHUTDOWN_EVENT: The peer has sent a SHUTDOWN. No further data should be sent on this socket.

SCTP_ADAPTION_INDICATION: This notification holds the peers indicated adaption layer. Please see Section 5.3.1.6.

SCTP_PARTIAL_DELIVERY_EVENT: This notification is used to tell a receiver that the partial delivery has been aborted. This may indicate the association is about to be aborted. Please see Section 5.3.1.7

All standard values for sn_type are greater than 2^15. Values from 2^15 and down are reserved.

sn_flags: 16 bits (unsigned integer)

These are notification-specific flags.

sn_length: 32 bits (unsigned integer)

This is the length of the whole sctp_notification structure including the sn_type, sn_flags, and sn_length fields.

5.3.1.1 SCTP_ASSOC_CHANGE

Communication notifications inform the ULP that an SCTP association has either begun or ended. The identifier for a new association is provided by this notification. The notification information has the following format:

```
struct sctp_assoc_change {
   uint16_t sac_type;
   uint16_t sac_flags;
   uint32_t sac_length;
   uint16_t sac_state;
   uint16_t sac_error;
   uint16_t sac_outbound_streams;
   uint16_t sac_inbound_streams;
   sctp_assoc_t sac_assoc_id;
   uint8_t  sac_info[0];
};
```

sac_type:

It should be SCTP_ASSOC_CHANGE.

sac_flags: 16 bits (unsigned integer)

Currently unused.

sac_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header.

sac_state: 16 bits (signed integer)

This field holds one of a number of values that communicate the event that happened to the association. They include:

Event Name	Description
SCTP_COMM_UP	A new association is now ready and data may be exchanged with this peer.
SCTP_COMM_LOST	The association has failed. The association is now in the closed state. If SEND FAILED notifications are turned on, a SCTP_COMM_LOST is followed by a series of SCTP_SEND_FAILED events, one for each outstanding message.
SCTP_RESTART	SCTP has detected that the peer has restarted.
SCTP_SHUTDOWN_COMP	The association has gracefully closed.
SCTP_CANT_STR_ASSOC	The association failed to setup. If non blocking mode is set and data was sent (in the udp mode), a SCTP_CANT_STR_ASSOC is followed by a series of SCTP_SEND_FAILED events, one for each outstanding message.

sac_error: 16 bits (signed integer)

If the state was reached due to a error condition (e.g. SCTP_COMM_LOST) any relevant error information is available in this field. This corresponds to the protocol error codes defined in RFC2960 [8].

sac_outbound_streams: 16 bits (unsigned integer)

sac_inbound_streams: 16 bits (unsigned integer)

The maximum number of streams allowed in each direction are available in sac_outbound_streams and sac_inbound streams.

sac_assoc_id: sizeof (sctp_assoc_t)

The association id field, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

sac_info: variable

If the sac_state is SCTP_COMM_LOST and an ABORT chunk was received for this association, sac_info[] contains the complete ABORT chunk as defined in the SCTP specification RFC2960 [8] section 3.3.7.

5.3.1.2 SCTP_PEER_ADDR_CHANGE

When a destination address on a multi-homed peer encounters a change an interface details event is sent. The information has the following structure:

```
    struct sctp_paddr_change {
       uint16_t spc_type;
       uint16_t spc_flags;
       uint32_t spc_length;
       struct sockaddr_storage spc_aaddr;
       int spc_state;
       int spc_error;
       sctp_assoc_t spc_assoc_id;
     }
```

spc_type:

It should be SCTP_PEER_ADDR_CHANGE.

spc_flags: 16 bits (unsigned integer)

Currently unused.

spc_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header.

spc_aaddr: sizeof (struct sockaddr_storage)

The affected address field, holds the remote peer's address that is encountering the change of state.

spc_state: 32 bits (signed integer)

This field holds one of a number of values that communicate the event that happened to the address. They include:

Event Name	Description
SCTP_ADDR_AVAILABLE	This address is now reachable.
SCTP_ADDR_UNREACHABLE	The address specified can no longer be reached. Any data sent to this address is rerouted to an alternate until this address becomes reachable.
SCTP_ADDR_REMOVED	The address is no longer part of the association.
SCTP_ADDR_ADDED	The address is now part of the association.
SCTP_ADDR_MADE_PRIM	This address has now been made to be the primary destination address.

spc_error: 32 bits (signed integer)

If the state was reached due to any error condition (e.g. SCTP_ADDR_UNREACHABLE) any relevant error information is available in this field.

spc_assoc_id: sizeof (sctp_assoc_t)

The association id field, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

5.3.1.3 SCTP_REMOTE_ERROR

A remote peer may send an Operational Error message to its peer. This message indicates a variety of error conditions on an association. The entire ERROR chunk as it appears on the wire is included in a SCTP_REMOTE_ERROR event. Please refer to the SCTP specification RFC2960 [8] and any extensions for a list of possible error formats. SCTP error notifications have the format:

```
struct sctp_remote_error {
    uint16_t sre_type;
    uint16_t sre_flags;
    uint32_t sre_length;
    uint16_t sre_error;
    sctp_assoc_t sre_assoc_id;
```

```
     uint8_t sre_data[0];
};
```

sre_type:

It should be SCTP_REMOTE_ERROR.

sre_flags: 16 bits (unsigned integer)

Currently unused.

sre_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header and the contents of sre_data.

sre_error: 16 bits (unsigned integer)

This value represents one of the Operational Error causes defined in the SCTP specification, in network byte order.

sre_assoc_id: sizeof (sctp_assoc_t)

The association id field, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

sre_data: variable

This contains the ERROR chunk as defined in the SCTP specification RFC2960 [8] section 3.3.10.

5.3.1.4 SCTP_SEND_FAILED

If SCTP cannot deliver a message it may return the message as a notification.

```
struct sctp_send_failed {
   uint16_t ssf_type;
   uint16_t ssf_flags;
   uint32_t ssf_length;
   uint32_t ssf_error;
   struct sctp_sndrcvinfo ssf_info;
   sctp_assoc_t ssf_assoc_id;
   uint8_t ssf_data[0];
};
```

ssf_type:

It should be SCTP_SEND_FAILED.

The flag value will take one of the following values

SCTP_DATA_UNSENT - Indicates that the data was never put on the wire.

SCTP_DATA_SENT - Indicates that the data was put on the wire. Note that this does not necessarily mean that the data was (or was not) successfully delivered.

ssf_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header and the payload in ssf_data.

ssf_error: 16 bits (unsigned integer)

This value represents the reason why the send failed, and if set, will be a SCTP protocol error code as defined in RFC2960 [8] section 3.3.10.

ssf_info: sizeof (struct sctp_sndrcvinfo)

The original send information associated with the undelivered message.

ssf_assoc_id: sizeof (sctp_assoc_t)

The association id field, sf_assoc_id, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

ssf_data: variable length

The undelivered message, exactly as delivered by the caller to the original send*() call.

5.3.1.5 SCTP_SHUTDOWN_EVENT

When a peer sends a SHUTDOWN, SCTP delivers this notification to inform the application that it should cease sending data.

```
struct sctp_shutdown_event {
    uint16_t sse_type;
    uint16_t sse_flags;
    uint32_t sse_length;
    sctp_assoc_t sse_assoc_id;
};
```

sse_type

It should be SCTP_SHUTDOWN_EVENT

sse_flags: 16 bits (unsigned integer)

Currently unused.

sse_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header. It will generally be sizeof (struct sctp_shutdown_event).

sse_flags: 16 bits (unsigned integer)

Currently unused.

sse_assoc_id: sizeof (sctp_assoc_t)

The association id field, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

5.3.1.6 SCTP_ADAPTION_INDICATION

When a peer sends a Adaption Layer Indication parameter , SCTP delivers this notification to inform the application that of the peers requested adaption layer.

```
struct sctp_adaption_event {
    uint16_t sai_type;
    uint16_t sai_flags;
    uint32_t sai_length;
    uint32_t sai_adaption_ind;
    sctp_assoc_t sai_assoc_id;
};
```

sai_type

It should be SCTP_ADAPTION_INDICATION

sai_flags: 16 bits (unsigned integer)

Currently unused.

sai_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header. It will generally be sizeof (struct sctp_adaption_event).

sai_adaption_ind: 32 bits (unsigned integer)

This field holds the bit array sent by the peer in the adaption layer indication parameter. The bits are in network byte order.

sai_assoc_id: sizeof (sctp_assoc_t)

The association id field, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

5.3.1.7 SCTP_PARTIAL_DELIVERY_EVENT

When a receiver is engaged in a partial delivery of a message this notification will be used to indicate various events.

```
struct sctp_pdapi_event {
    uint16_t pdapi_type;
    uint16_t pdapi_flags;
    uint32_t pdapi_length;
    uint32_t pdapi_indication;
    sctp_assoc_t pdapi_assoc_id;
};
```

pdapi_type

It should be SCTP_PARTIAL_DELIVERY_EVENT

pdapi_flags: 16 bits (unsigned integer)

Currently unused.

pdapi_length: 32 bits (unsigned integer)

This field is the total length of the notification data, including the notification header. It will generally be sizeof (struct sctp_pdapi_event).

pdapi_indication: 32 bits (unsigned integer)

This field holds the indication being sent to the application possible values include:

SCTP_PARTIAL_DELIVERY_ABORTED

pdapi_assoc_id: sizeof (sctp_assoc_t)

The association id field, holds the identifier for the association. All notifications for a given association have the same association identifier. For one-to-one style socket, this field is ignored.

5.4 Ancillary Data Considerations and Semantics

Programming with ancillary socket data contains some subtleties and pitfalls, which are discussed below.

5.4.1 Multiple Items and Ordering

Multiple ancillary data items may be included in any call to sendmsg() or recvmsg(); these may include multiple SCTP or non-SCTP items, or both.

The ordering of ancillary data items (either by SCTP or another protocol) is not significant and is implementation-dependent, so applications must not depend on any ordering.

SCTP_SNDRCV items must always correspond to the data in the msghdr's msg_iov member. There can be only a single SCTP_SNDRCV info for each sendmsg() or recvmsg() call.

5.4.2 Accessing and Manipulating Ancillary Data

Applications can infer the presence of data or ancillary data by examining the msg_iovlen and msg_controllen msghdr members, respectively.

Implementations may have different padding requirements for ancillary data, so portable applications should make use of the macros CMSG_FIRSTHDR, CMSG_NXTHDR, CMSG_DATA, CMSG_SPACE, and CMSG_LEN. See RFC2292 [6] and your SCTP implementation's documentation for more information. Following is an example, from RFC2292 [6], demonstrating the use of these macros to access ancillary data:

```
struct msghdr   msg;
struct cmsghdr  *cmsgptr;

/* fill in msg */

/* call recvmsg() */

for (cmsgptr = CMSG_FIRSTHDR(&msg); cmsgptr != NULL;
     cmsgptr = CMSG_NXTHDR(&msg, cmsgptr)) {
     if (cmsgptr->cmsg_level == ... && cmsgptr->cmsg_type == ... ) {
        u_char  *ptr;

        ptr = CMSG_DATA(cmsgptr);
        /* process data pointed to by ptr */
     }
}
```

5.4.3 Control Message Buffer Sizing

The information conveyed via SCTP_SNDRCV events will often be fundamental to the correct and sane operation of the sockets application. This is particularly true of the one-to-many semantics, but also of the one-ton-one semantics. For example, if an application needs to send and receive data on different SCTP streams, SCTP_SNDRCV events are indispensable.

Given that some ancillary data is critical, and that multiple ancillary data items may appear in any order, applications should be carefully written to always provide a large enough buffer to contain all possible ancillary data that can be presented by recvmsg(). If the buffer is too small, and crucial data is truncated, it may pose a fatal error condition.

Thus it is essential that applications be able to deterministically calculate the maximum required buffer size to pass to recvmsg(). One constraint imposed on this specification that makes this possible is that all ancillary data definitions are of a fixed length. One way to calculate the maximum required buffer size might be to take the sum the sizes of all enabled ancillary data item structures, as calculated by CMSG_SPACE. For example, if we enabled SCTP_SNDRCV_INFO and IPV6_RECVPKTINFO RFC2292 [6], we would calculate and allocate the buffer size as follows:

```
size_t total;
void *buf;

total = CMSG_SPACE(sizeof (struct sctp_sndrcvinfo)) +
        CMSG_SPACE(sizeof (struct in6_pktinfo));

buf = malloc(total);
```

We could then use this buffer for msg_control on each call to recvmsg() and be assured that we would not lose any ancillary data to truncation.

6. Common Operations for Both Styles

6.1 send(), recv(), sendto(), recvfrom()

Applications can use send() and sendto() to transmit data to the peer of an SCTP endpoint. recv() and recvfrom() can be used to receive data from the peer.

The syntax is:

```
ssize_t send(int sd, const void *msg, size_t len, int flags);
ssize_t sendto(int sd, const void *msg, size_t len, int flags,
               const struct sockaddr *to, socklen_t tolen);
ssize_t recv(int sd, void *buf, size_t len, int flags);
ssize_t recvfrom(int sd, void *buf, size_t len, int flags,
                 struct sockaddr *from, socklen_t *fromlen);
```

```
  sd      - the socket descriptor of an SCTP endpoint.
  msg     - the message to be sent.
  len     - the size of the message or the size of buffer.
  to      - one of the peer addresses of the association to be
            used to send the message.
  tolen   - the size of the address.
  buf     - the buffer to store a received message.
  from    - the buffer to store the peer address used to send the
            received message.
  fromlen - the size of the from address
  flags   - (described below).
```

These calls give access to only basic SCTP protocol features. If either peer in the association uses multiple streams, or sends unordered data these calls will usually be inadequate, and may deliver the data in unpredictable ways.

SCTP has the concept of multiple streams in one association. The above calls do not allow the caller to specify on which stream a message should be sent. The system uses stream 0 as the default stream for send() and sendto(). recv() and recvfrom() return data from any stream, but the caller can not distinguish the different streams. This may result in data seeming to arrive out of order. Similarly, if a data chunk is sent unordered, recv() and recvfrom() provide no indication.

SCTP is message based. The msg buffer above in send() and sendto() is considered to be a single message. This means that if the caller wants to send a message which is composed by several buffers, the caller needs to combine them before calling send() or sendto(). Alternately, the caller can use sendmsg() to do that without

combining them. recv() and recvfrom() cannot distinguish message boundaries.

In receiving, if the buffer supplied is not large enough to hold a complete message, the receive call acts like a stream socket and returns as much data as will fit in the buffer.

Note, the send() and recv() calls may not be used for a one-to-many style socket.

Note, if an application calls a send function with no user data and no ancillary data the SCTP implementation should reject the request with an appropriate error message. An implementation is NOT allowed to send a Data chunk with no user data RFC2960 [8].

6.2 setsockopt(), getsockopt()

Applications use setsockopt() and getsockopt() to set or retrieve socket options. Socket options are used to change the default behavior of sockets calls. They are described in Section 7

The syntax is:

```
ret = getsockopt(int sd, int level, int optname, void *optval,
                 socklen_t *optlen);
ret = setsockopt(int sd, int level, int optname, const void *optval,
                 socklen_t optlen);
```

```
  sd      - the socket descript.
  level   - set to IPPROTO_SCTP for all SCTP options.
  optname - the option name.
  optval  - the buffer to store the value of the option.
  optlen  - the size of the buffer (or the length of  the option
            returned).
```

6.3 read() and write()

Applications can use read() and write() to send and receive data to and from peer. They have the same semantics as send() and recv() except that the flags parameter cannot be used.

Note, these calls, when used in the one-to-many style, may only be used with branched off socket descriptors (see Section 8.2).

6.4 getsockname()

Applications use getsockname() to retrieve the locally-bound socket address of the specified socket. This is especially useful if the caller let SCTP chose a local port. This call is for where the endpoint is not multi-homed. It does not work well with multi-homed sockets. See Section 8.5 for a multi-homed version of the call.

The syntax is:

```
int getsockname(int sd, struct sockaddr *address,
                socklen_t *len);
```

```
  sd      - the socket descriptor to be queried.

  address - On return, one locally bound address (chosen by
            the SCTP stack) is stored in this buffer. If the
            socket is an IPv4 socket, the address will be IPv4.
            If the socket is an IPv6 socket, the address will
            be either an IPv6 or IPv4 address.

  len     - The caller should set the length of address here.
            On return, this is set to the length of the returned
            address.
```

If the actual length of the address is greater than the length of the supplied sockaddr structure, the stored address will be truncated.

If the socket has not been bound to a local name, the value stored in the object pointed to by address is unspecified.

7. Socket Options

The following sub-section describes various SCTP level socket options that are common to both styles. SCTP associations can be multi-homed. Therefore, certain option parameters include a sockaddr_storage structure to select which peer address the option should be applied to.

For the one-to-many style sockets, an sctp_assoc_t structure (association ID) is used to identify the the association instance that the operation affects. So it must be set when using this style.

For the one-to-one style sockets and branched off one-to-many style sockets (see Section 8.2) this association ID parameter is ignored.

Note that socket or IP level options are set or retrieved per socket. This means that for one-to-many style sockets, those options will be applied to all associations belonging to the socket. And for one-to-one style, those options will be applied to all peer addresses of the association controlled by the socket. Applications should be very careful in setting those options.

For some IP stacks getsockopt() is read-only; so a new interface will be needed when information must be passed both in to and out of the SCTP stack. The syntax for sctp_opt_info() is,

```
int sctp_opt_info(int sd,
                  sctp_assoc_t id,
                  int opt,
                  void *arg,
                  socklen_t *size);
```

The sctp_opt_info() call is a replacement for getsockopt() only and will not set any options associated with the specified socket. A setsockopt() must be used to set any writeable option.

For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored.

opt specifies which SCTP socket option to get. It can get any socket option currently supported that requests information (either read/write options or read only) such as:

SCTP_RTOINFO
SCTP_ASSOCINFO
SCTP_DEFAULT_SEND_PARAM
SCTP_GET_PEER_ADDR_INFO
SCTP_PRIMARY_ADDR
SCTP_PEER_ADDR_PARAMS
SCTP_STATUS
SCTP_AUTH_CHUNKS
SCTP_AUTH_SECRET

arg is an option-specific structure buffer provided by the caller. See Section 8.5) subsections for more information on these options and option-specific structures.

sctp_opt_info() returns 0 on success, or on failure returns -1 and sets errno to the appropriate error code.

All options that support specific settings on an association by filling in either an association id variable or a sockaddr_storage SHOULD also support setting of the same value for the entire endpoint (i.e. future associations). To accomplish this the following logic is used when setting one of these options:

a) If an address is specified via a sockaddr_storage that is included in the structure, the address is used to lookup the association and the settings are applied to the specific address (if appropriate) or to the entire association.
b) If an association identification is filled in but not a sockaddr_storage (if present), the association is found using the association identification and the settings should be applied to the entire association (since a specific address is not specified). Note this also applies to options that hold an association identification in their structure but do not have a sockaddr_storage field.
c) If neither the sockaddr_storage or association identification is set i.e. the sockaddr_storage is set to all 0's (INADDR_ANY) and the association identification is 0, the settings are a default and to be applied to the endpoint (all future associations).

7.1 Read / Write Options

7.1.1 Retransmission Timeout Parameters (SCTP_RTOINFO)

The protocol parameters used to initialize and bound retransmission timeout (RTO) are tunable. See RFC2960 [8] for more information on how these parameters are used in RTO calculation.

The following structure is used to access and modify these

parameters:

```
struct sctp_rtoinfo {
    sctp_assoc_t          srto_assoc_id;
    uint32_t              srto_initial;
    uint32_t              srto_max;
    uint32_t              srto_min;
};
  srto_initial    - This contains the initial RTO value.
  srto_max and srto_min - These contain the maximum and minimum bounds
                        for all RTOs.
  srto_assoc_id   - (one-to-many style socket) This is filled in
                    the application, and identifies the association
                    for this query. If this parameter is '0'
                    (on a one-to-many style socket), then the change
                    effects the entire endpoint.
```

All parameters are time values, in milliseconds. A value of 0, when modifying the parameters, indicates that the current value should not be changed.

To access or modify these parameters, the application should call getsockopt or setsockopt() respectively with the option name SCTP_RTOINFO.

7.1.2 Association Parameters (SCTP_ASSOCINFO)

This option is used to both examine and set various association and endpoint parameters.

See RFC2960 [8] for more information on how this parameter is used. The peer address parameter is ignored for one-to-one style socket.

The following structure is used to access and modify this parameters:

```
struct sctp_assocparams {
   sctp_assoc_t    sasoc_assoc_id;
   uint16_t        sasoc_asocmaxrxt;
   uint16_t        sasoc_number_peer_destinations;
   uint32_t        sasoc_peer_rwnd;
   uint32_t        sasoc_local_rwnd;
   uint32_t        sasoc_cookie_life;
};
```

sasoc_asocmaxrxt - This contains the maximum retransmission attempts to make for the association.

sasoc_number_peer_destinations - This is the number of destination addresses that the peer has.

sasoc_peer_rwnd - This holds the current value of the peers rwnd (reported in the last SACK) minus any outstanding data (i.e. data inflight).

sasoc_local_rwnd - This holds the last reported rwnd that was sent to the peer.

sasoc_cookie_life - This is the associations cookie life value used when issuing cookies.

sasoc_assoc_id - (one-to-many style socket) This is filled in the application, and identifies the association for this query.

This information may be examined for either the endpoint or a specific association. To examine a endpoints default parameters the association id (sasoc_assoc_id) should must be set to the value '0'. The values of the sasoc_peer_rwnd is meaningless when examining endpoint information.

All parameters are time values, in milliseconds. A value of 0, when modifying the parameters, indicates that the current value should not be changed.

The values of the sasoc_asocmaxrxt and sasoc_cookie_life may be set on either an endpoint or association basis. The rwnd and destination counts (sasoc_number_peer_destinations, sasoc_peer_rwnd,sasoc_local_rwnd) are NOT settable and any value placed in these is ignored.

To access or modify these parameters, the application should call getsockopt or setsockopt() respectively with the option name SCTP_ASSOCINFO.

The maximum number of retransmissions before an address is considered unreachable is also tunable, but is address-specific, so it is covered in a separate option. If an application attempts to set the value of the association maximum retransmission parameter to more

than the sum of all maximum retransmission parameters, setsockopt() shall return an error. The reason for this, from RFC2960 [8] section 8.2:

Note: When configuring the SCTP endpoint, the user should avoid having the value of 'Association.Max.Retrans' larger than the summation of the 'Path.Max.Retrans' of all the destination addresses for the remote endpoint. Otherwise, all the destination addresses may become inactive while the endpoint still considers the peer endpoint reachable.

7.1.3 Initialization Parameters (SCTP_INITMSG)

Applications can specify protocol parameters for the default association initialization. The structure used to access and modify these parameters is defined in Section 5.2.1). The option name argument to setsockopt() and getsockopt() is SCTP_INITMSG.

Setting initialization parameters is effective only on an unconnected socket (for one-to-many style sockets only future associations are effected by the change). With one-to-one style sockets, this option is inherited by sockets derived from a listener socket.

7.1.4 SO_LINGER

An application using the one-to-one style socket can use this option to perform the SCTP ABORT primitive. The linger option structure is:

```
struct  linger {
    int     l_onoff;                /* option on/off */
    int     l_linger;               /* linger time */
};
```

To enable the option, set l_onoff to 1. If the l_linger value is set to 0, calling close() is the same as the ABORT primitive. If the value is set to a negative value, the setsockopt() call will return an error. If the value is set to a positive value linger_time, the close() can be blocked for at most linger_time ms. If the graceful shutdown phase does not finish during this period, close() will return but the graceful shutdown phase continues in the system.

Note, this is a socket level option NOT an SCTP level option. So when setting SO_LINGER you must specify a level of SOL_SOCKET in the setsockopt() call.

7.1.5 SCTP_NODELAY

Turn on/off any Nagle-like algorithm. This means that packets are

generally sent as soon as possible and no unnecessary delays are introduced, at the cost of more packets in the network. Expects an integer boolean flag.

7.1.6 SO_RCVBUF

Sets receive buffer size in octets. For SCTP one-to-one style sockets, this controls the receiver window size. For one-to-many style sockets the meaning depends on the constant HAVE_SCTP_MULTIBUF (see Section 3.4). If the implementation defines HAVE_SCTP_MULTIBUF as 1, this controls the receiver window size for each association bound to the socket descriptor. If the implementation defines HAVE_SCTP_MULTIBUF as 0, this controls the size of the single receive buffer for the whole socket. The call expects an integer.

7.1.7 SO_SNDBUF

Sets send buffer size. For SCTP one-to-one style sockets, this controls the amount of data SCTP may have waiting in internal buffers to be sent. This option therefore bounds the maximum size of data that can be sent in a single send call. For one-to-many style sockets, the effect is the same, except that it applies to one or all associations (see Section 3.4) bound to the socket descriptor used in the setsockopt() or getsockopt() call. The option applies to each association's window size separately. The call expects an integer.

7.1.8 Automatic Close of associations (SCTP_AUTOCLOSE)

This socket option is applicable to the one-to-many style socket only. When set it will cause associations that are idle for more than the specified number of seconds to automatically close. An association being idle is defined as an association that has NOT sent or received user data. The special value of '0' indicates that no automatic close of any associations should be performed, this is the default value. The option expects an integer defining the number of seconds of idle time before an association is closed.

An application using this option should enable receiving the association change notification. This is the only mechanism an application is informed about the closing of an association. After an association is closed, the association ID assigned to it can be reused. An application should be aware of this to avoid the possible problem of sending data to an incorrect peer end point.

7.1.9 Set Peer Primary Address (SCTP_SET_PEER_PRIMARY_ADDR)

Requests that the peer mark the enclosed address as the association

primary. The enclosed address must be one of the association's locally bound addresses. The following structure is used to make a set primary request:

```
struct sctp_setpeerprim {
    sctp_assoc_t            sspp_assoc_id;
    struct sockaddr_storage sspp_addr;
};
```

```
  sspp_addr           The address to set as primary
  sspp_assoc_id       (one-to-many style socket) This is filled in by the
                      application, and identifies the association
                      for this request.
```

This functionality is optional. Implementations that do not support this functionality should return EOPNOTSUPP.

7.1.10 Set Primary Address (SCTP_PRIMARY_ADDR)

Requests that the local SCTP stack use the enclosed peer address as the association primary. The enclosed address must be one of the association peer's addresses. The following structure is used to make a set peer primary request:

```
struct sctp_setprim {
    sctp_assoc_t            ssp_assoc_id;
    struct sockaddr_storage ssp_addr;
};
  ssp_addr            The address to set as primary
  ssp_assoc_id        (one-to-many style socket) This is filled in by the
                      application, and identifies the association
                      for this request.
```

7.1.11 Set Adaption Layer Indicator (SCTP_ADAPTION_LAYER)

Requests that the local endpoint set the specified Adaption Layer Indication parameter for all future INIT and INIT-ACK exchanges.

```
    struct sctp_setadaption {
        uint32_t   ssb_adaption_ind;
    };

    ssb_adaption_ind    The adaption layer indicator that will be included
                        in any outgoing Adaption Layer Indication
                        parameter.
```

7.1.12 Enable/Disable message fragmentation (SCTP_DISABLE_FRAGMENTS)

This option is a on/off flag and is passed an integer where a non-zero is on and a zero is off. If enabled no SCTP message fragmentation will be performed. Instead if a message being sent exceeds the current PMTU size, the message will NOT be sent and instead a error will be indicated to the user.

7.1.13 Peer Address Parameters (SCTP_PEER_ADDR_PARAMS)

Applications can enable or disable heartbeats for any peer address of an association, modify an address's heartbeat interval, force a heartbeat to be sent immediately, and adjust the address's maximum number of retransmissions sent before an address is considered unreachable. The following structure is used to access and modify an address's parameters:

```
struct sctp_paddrparams {
    sctp_assoc_t            spp_assoc_id;
    struct sockaddr_storage spp_address;
    uint32_t                spp_hbinterval;
    uint16_t                spp_pathmaxrxt;
    uint32_t                spp_pathmtu;
    uint32_t                spp_sackdelay;
    uint32_t                spp_flags;
};
```

```
  spp_assoc_id    - (one-to-many style socket) This is filled in the
                    application, and identifies the association for
                    this query.
  spp_address     - This specifies which address is of interest.
  spp_hbinterval  - This contains the value of the heartbeat interval,
                    in milliseconds.  If a  value of zero
                    is present in this field then no changes are to
                    be made to this parameter.
  spp_pathmaxrxt  - This contains the maximum number of
                    retransmissions before this address shall be
                    considered unreachable. If a  value of zero
                    is present in this field then no changes are to
                    be made to this parameter.
  spp_pathmtu     - When Path MTU discovery is disabled the value
                    specified here will be the "fixed" path mtu.
                    Note that if the spp_address field is empty
                    then all associations on this address will
                    have this fixed path mtu set upon them.

  spp_sackdelay   - When delayed sack is enabled, this value specifies
                    the number of milliseconds that sacks will be delayed
```

for. This value will apply to all addresses of an association if the spp_address field is empty. Note also, that if delayed sack is enabled and this value is set to 0, no change is made to the last recorded delayed sack timer value.

spp_flags - These flags are used to control various features on an association. The flag field may contain zero or more of the following options.

SPP_HB_ENABLED - Enable heartbeats on the specified address. Note that if the address field is empty all addresses for the association have heartbeats enabled upon them.

SPP_HB_DISABLED - Disable heartbeats on the speicifed address. Note that if the address field is empty all addresses for the association will have their heartbeats disabled. Note also that SPP_HB_ENABLED and SPP_HB_DISABLED are mutually exclusive, only one of these two should be specified. Enabling both fields will have undetermined results.

SPP_PMTUD_ENABLED - This field will enable PMTU discovery upon the specified address. Note that if the address feild is empty then all addresses on the association are effected.

SPP_PMTUD_DISABLED - This field will disable PMTU discovery upon the specified address. Note that if the address feild is empty then all addresses on the association are effected. Not also that SPP_PMTUD_ENABLE and SPP_PMTUD_DISABLE are mutually exclusive. Enabling both will have undetermined results.

SPP_SACKDELAY_ENABLED - Setting this flag turns on delayed sack. The time specified in spp_sackdelay is used to specify the sack delay for this address. Note that if spp_address is empty then all addresses will enable delayed sack and take on the sack delay value specified in spp_sackdelay.

SPP_SACKDELAY_DISABLED - Setting this flag turns off delayed sack. If the spp_address field is blank then delayed sack is disabled for the entire association.

Note

also that this field is mutually exclusive to SPP_SACKDELAY_ENABLED, setting both will have undefined results.

To read or modify these parameters, the application should call sctp_opt_info() with the SCTP_PEER_ADDR_PARAMS option.

7.1.14 Set default send parameters (SCTP_DEFAULT_SEND_PARAM)

Applications that wish to use the sendto() system call may wish to specify a default set of parameters that would normally be supplied through the inclusion of ancillary data. This socket option allows such an application to set the default sctp_sndrcvinfo structure. The application that wishes to use this socket option simply passes in to this call the sctp_sndrcvinfo structure defined in Section 5.2.2) The input parameters accepted by this call include sinfo_stream, sinfo_flags, sinfo_ppid, sinfo_context, sinfo_timetolive. The user must set the sinfo_assoc_id field to identify the association to affect if the caller is using the one-to-many style.

7.1.15 Set notification and ancillary events (SCTP_EVENTS)

This socket option is used to specify various notifications and ancillary data the user wishes to receive. Please see Section 7.3) for a full description of this option and its usage.

7.1.16 Set/clear IPv4 mapped addresses (SCTP_I_WANT_MAPPED_V4_ADDR)

This socket option is a boolean flag which turns on or off mapped V4 addresses. If this option is turned on and the socket is type PF_INET6, then IPv4 addresses will be mapped to V6 representation. If this option is turned off, then no mapping will be done of V4 addresses and a user will receive both PF_INET6 and PF_INET type addresses on the socket.

By default this option is turned on and expects an integer to be passed where non-zero turns on the option and zero turns off the option.

7.1.17 Set the maximum fragmentation size (SCTP_MAXSEG)

This socket option specifies the maximum size to put in any outgoing SCTP DATA chunk. If a message is larger than this size it will be fragmented by SCTP into the specified size. Note that the underlying SCTP implementation may fragment into smaller sized chunks when the PMTU of the underlying association is smaller than the value set by the user. The option expects an integer.

The default value for this option is '0' which indicates the user is NOT limiting fragmentation and only the PMTU will effect SCTP's choice of DATA chunk size.

7.1.18 Set/Get the list of chunks that must be authenticated (SCTP_AUTH_CHUNKS)

This options gets or sets a list of chunks that the user is requesting to be received only in an authenticated way. Changes to this list will only effect associations that have not been formed.

```
struct sctp_authchunks {
    uint8_t                 sauth_chunks[];
};
```

sauth_chunks - This parameter contains an array of chunks that the user is requesting to be authenticated.

7.1.19 Set/Get the current authentication shared secret (SCTP_AUTH_SECRET)

This option will get or set the shared secret to be used with any authentication parameters.

```
struct sctp_authsecrets {
    sctp_assoc_t            sca_assoc_id;
    uint8_t                 sca_secret[];
};
```

sca_assoc_id - This parameter, if non-zero, indicates what association that the shared secret is being set upon. Note that if this element contains zero, then the secret is set upon the endpoint and all future associations will use this secret (if not changed by subsequent calls to SCTP_AUTH_SECRET).

sca_secret - This parameter contains an array of bytes that is to be used by the endpoint (or association) as the shared secret.

7.1.20 Get the list of chunks that peer requires to be authenticated (SCTP_PEER_AUTH_CHUNKS)

This options gets a list of chunks for a specified association that the peer requires to be authenticated. The requesting to be received only in an authenticated way. Changes to this list will only effect associations that have not been formed.

```
struct sctp_authchunks {
    sctp_assoc_t            gpauth_assoc_id;
    uint8_t                 gpauth_chunks[];
};
```

```
  sca_assoc_id    - This parameter, indicates for which association the
                    user is requesting the list of peer authenticated
                    chunks.

  gputh_chunks    - This parameter contains an array of chunks
                    that the peer is requesting to be authenticated.
```

7.2 Read-Only Options

7.2.1 Association Status (SCTP_STATUS)

Applications can retrieve current status information about an association, including association state, peer receiver window size, number of unacked data chunks, and number of data chunks pending receipt. This information is read-only. The following structure is used to access this information:

```
struct sctp_status {
    sctp_assoc_t    sstat_assoc_id;
    int32_t         sstat_state;
    uint32_t        sstat_rwnd;
    uint16_t        sstat_unackdata;
    uint16_t        sstat_penddata;
    uint16_t        sstat_instrms;
    uint16_t        sstat_outstrms;
    uint32_t        sstat_fragmentation_point;
    struct sctp_paddrinfo sstat_primary;
};
```

```
  sstat_state    - This contains the association's current state one
                   of the following values:

                     SCTP_CLOSED
```

```
                        SCTP_BOUND
                        SCTP_LISTEN
                        SCTP_COOKIE_WAIT
                        SCTP_COOKIE_ECHOED
                        SCTP_ESTABLISHED
                        SCTP_SHUTDOWN_PENDING
                        SCTP_SHUTDOWN_SENT
                        SCTP_SHUTDOWN_RECEIVED
                        SCTP_SHUTDOWN_ACK_SENT

     sstat_rwnd     - This contains the association  peer's current
                      receiver window size.
     sstat_unackdata - This is the number of unacked data chunks.
     sstat_penddata  - This is the number of data chunks pending receipt.
     sstat_primary   - This is information on the current primary peer
                       address.
     sstat_assoc_id  - (one-to-many style socket) This holds the an
                       identifier for the association.  All
                       notifications for a given association
                       have the same association identifier.

     sstat_instrms   - The number of streams that the peer will
                       be using inbound.

     sstat_outstrms  - The number of streams that the endpoint is
                       allowed to use outbound.

     sstat_fragmentation_point - The size at which SCTP fragmentation
                                 will occur.
```

To access these status values, the application calls getsockopt() with the option name SCTP_STATUS. The sstat_assoc_id parameter is ignored for one-to-one style socket.

7.2.2 Peer Address Information (SCTP_GET_PEER_ADDR_INFO)

Applications can retrieve information about a specific peer address of an association, including its reachability state, congestion window, and retransmission timer values. This information is read-only. The following structure is used to access this information:

```
struct sctp_paddrinfo {
    sctp_assoc_t    spinfo_assoc_id;
    struct sockaddr_storage spinfo_address;
    int32_t         spinfo_state;
    uint32_t        spinfo_cwnd;
    uint32_t        spinfo_srtt;
    uint32_t        spinfo_rto;
    uint32_t        spinfo_mtu;
};
```

spinfo_address   - This is filled in the application, and contains the peer address of interest.

On return from getsockopt():

spinfo_state   - This contains the peer addresses's state (either SCTP_ACTIVE or SCTP_INACTIVE and possibly the modifer SCTP_UNCONFIRMED)

spinfo_cwnd   - This contains the peer addresses's current congestion window.

spinfo_srtt   - This contains the peer addresses's current smoothed round-trip time calculation in milliseconds.

spinfo_rto   - This contains the peer addresses's current retransmission timeout value in milliseconds.

spinfo_mtu   - The current P-MTU of this address.

spinfo_assoc_id - (one-to-many style socket) This is filled in the application, and identifies the association for this query.

To retrieve this information, use sctp_opt_info() with the SCTP_GET_PEER_ADDR_INFO options.

7.3 Ancillary Data and Notification Interest Options

Applications can receive per-message ancillary information and notifications of certain SCTP events with recvmsg().

The following optional information is available to the application:

1. SCTP_SNDRCV (sctp_data_io_event): Per-message information (i.e. stream number, TSN, SSN, etc. described in Section 5.2.2)
2. SCTP_ASSOC_CHANGE (sctp_association_event): (described in Section 5.3.1.1)
3. SCTP_PEER_ADDR_CHANGE (sctp_address_event): (described in Section 5.3.1.2)
4. SCTP_SEND_FAILED (sctp_send_failure_event): (described in Section 5.3.1.4)

5. SCTP_REMOTE_ERROR (sctp_peer_error_event): (described in Section 5.3.1.3)
6. SCTP_SHUTDOWN_EVENT (sctp_shtudown_event): (described in Section 5.3.1.5)
7. SCTP_PARTIAL_DELIVERY_EVENT (sctp_partial_delivery_event): (described in Section 5.3.1.7)
8. SCTP_ADAPTION_INDICATION (sctp_adaption_layer_event): (described in Section 5.3.1.6)

To receive any ancillary data or notifications, first the application registers it's interest by calling the SCTP_EVENTS setsockopt() with the following structure.

```
struct sctp_event_subscribe{
    uint8_t sctp_data_io_event;
    uint8_t sctp_association_event;
    uint8_t sctp_address_event;
    uint8_t sctp_send_failure_event;
    uint8_t sctp_peer_error_event;
    uint8_t sctp_shutdown_event;
    uint8_t sctp_partial_delivery_event;
    uint8_t sctp_adaption_layer_event;
};
```

sctp_data_io_event - Setting this flag to 1 will cause the reception of SCTP_SNDRCV information on a per message basis. The application will need to use the recvmsg() interface so that it can receive the event information contained in the msg_control field. Please see Section 5.2 for further details. Setting the flag to 0 will disable reception of the message control information.

sctp_association_event - Setting this flag to 1 will enable the reception of association event notifications. Setting the flag to 0 will disable association event notifications. For more information on event notifications please see Section 5.3.

sctp_address_event - Setting this flag to 1 will enable the reception of address event notifications. Setting the flag to 0 will disable address event notifications. For more information on event notifications please see Section 5.3.

sctp_send_failure_event - Setting this flag to 1 will enable the reception of send failure event notifications. Setting the flag to 0 will disable send failure event notifications. For more information on event notifications please see Section 5.3.

sctp_peer_error_event - Setting this flag to 1 will enable the reception of peer error event notifications. Setting the flag to 0

will disable peer error event notifications. For more information on event notifications please see Section 5.3.

sctp_shutdown_event - Setting this flag to 1 will enable the reception of shutdown event notifications. Setting the flag to 0 will disable shutdown event notifications. For more information on event notifications please see Section 5.3.

sctp_partial_delivery_event - Setting this flag to 1 will enable the reception of partial delivery notifications. Setting the flag to 0 will disable partial delivery event notifications. For more information on event notifications please see Section 5.3.

sctp_adaption_layer_event - Setting this flag to 1 will enable the reception of adaption layer notifications. Setting the flag to 0 will disable adaption layer event notifications. For more information on event notifications please see Section 5.3.

An example where an application would like to receive data io events and association events but no others would be as follows:

```
{
   struct sctp_event_subscribe event;

   memset(&event,0,sizeof(event));

   event.sctp_data_io_event = 1;
   event.sctp_association_event = 1;

   setsockopt(fd, IPPROTO_SCTP, SCTP_EVENT, &event, sizeof(event));
}
```

Note that for one-to-many style SCTP sockets, the caller of recvmsg() receives ancillary data and notifications for ALL associations bound to the file descriptor. For one-to-one style SCTP sockets, the caller receives ancillary data and notifications for only the single association bound to the file descriptor.

By default both the one-to-one style and one-to-many style socket has all options off.

8. New Interfaces

Depending on the system, the following interface can be implemented as a system call or library function.

8.1 sctp_bindx()

The syntax of sctp_bindx() is,

```
int sctp_bindx(int sd, struct sockaddr *addrs, int addrcnt,
               int flags);
```

If sd is an IPv4 socket, the addresses passed must be IPv4 addresses. If the sd is an IPv6 socket, the addresses passed can either be IPv4 or IPv6 addresses.

A single address may be specified as INADDR_ANY or IN6ADDR_ANY, see Section 3.1.2 for this usage.

addrs is a pointer to an array of one or more socket addresses. Each address is contained in its appropriate structure. For an IPv6 socket, an array of sockaddr_in6 would be returned. For a IPv4 socket, an array of sockaddr_in would be returned. The caller specifies the number of addresses in the array with addrcnt. Note that the wildcard addresses cannot be used with this function, doing so will result in an error.

On success, sctp_bindx() returns 0. On failure, sctp_bindx() returns -1, and sets errno to the appropriate error code.

For SCTP, the port given in each socket address must be the same, or sctp_bindx() will fail, setting errno to EINVAL.

The flags parameter is formed from the bitwise OR of zero or more of the following currently defined flags:

SCTP_BINDX_ADD_ADDR

SCTP_BINDX_REM_ADDR

SCTP_BINDX_ADD_ADDR directs SCTP to add the given addresses to the association, and SCTP_BINDX_REM_ADDR directs SCTP to remove the given addresses from the association. The two flags are mutually exclusive; if both are given, sctp_bindx() will fail with EINVAL. A caller may not remove all addresses from an association; sctp_bindx() will reject such an attempt with EINVAL.

An application can use sctp_bindx(SCTP_BINDX_ADD_ADDR) to associate

additional addresses with an endpoint after calling bind(). Or use sctp_bindx(SCTP_BINDX_REM_ADDR) to remove some addresses a listening socket is associated with so that no new association accepted will be associated with those addresses. If the endpoint supports dynamic address a SCTP_BINDX_REM_ADDR or SCTP_BINDX_ADD_ADDR may cause a endpoint to send the appropriate message to the peer to change the peers address lists.

Adding and removing addresses from a connected association is optional functionality. Implementations that do not support this functionality should return EOPNOTSUPP.

8.2 Branched-off Association

After an association is established on a one-to-many style socket, the application may wish to branch off the association into a separate socket/file descriptor.

This is particularly desirable when, for instance, the application wishes to have a number of sporadic message senders/receivers remain under the original one-to-many style socket but branch off those associations carrying high volume data traffic into their own separate socket descriptors.

The application uses sctp_peeloff() call to branch off an association into a separate socket (Note the semantics are somewhat changed from the traditional one-to-one style accept() call). Note that the new socket is a one-to-one style socket. Thus it will be confined to operations allowed for a one-to-one style socket.

The syntax is:

```
new_sd = sctp_peeloff(int sd, sctp_assoc_t assoc_id);
```

the new socket descriptor representing the branched-off association.
the original one-to-many style socket descriptor returned from the socket() system call (see Section 3.1.1).
the specified identifier of the association that is to be branched off to a separate file descriptor (Note, in a traditional one-to-one style accept() call, this would be an out parameter, but for the one-to-many style call, this is an in parameter).

8.3 sctp_getpaddrs()

sctp_getpaddrs() returns all peer addresses in an association. The syntax is,

```
int sctp_getpaddrs(int sd, sctp_assoc_t id,
                   struct sockaddr **addrs);
```

On return, addrs will point to an array dynamically allocated sockaddr structures of the appropriate type for the socket type. The caller should use sctp_freepaddrs() to free the memory. Note that the in/out parameter addrs must not be NULL.

If sd is an IPv4 socket, the addresses returned will be all IPv4 addresses. If sd is an IPv6 socket, the addresses returned can be a mix of IPv4 or IPv6 addresses.

For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored.

On success, sctp_getpaddrs() returns the number of peer addresses in the association. If there is no association on this socket, sctp_getpaddrs() returns 0, and the value of *addrs is undefined. If an error occurs, sctp_getpaddrs() returns -1, and the value of *addrs is undefined.

8.4 sctp_freepaddrs()

sctp_freepaddrs() frees all resources allocated by sctp_getpaddrs(). Its syntax is,

```
void sctp_freepaddrs(struct sockaddr *addrs);
```

addrs is the array of peer addresses returned by sctp_getpaddrs().

8.5 sctp_getladdrs()

sctp_getladdrs() returns all locally bound address(es) on a socket. The syntax is,

```
int sctp_getladdrs(int sd, sctp_assoc_t id,
                   struct sockaddr **ss);
```

On return, addrs will point to a dynamically allocated array of sockaddr structures of the appropriate type for the socket type. The caller should use sctp_freeladdrs() to free the memory. Note that the in/out parameter addrs must not be NULL.

If sd is an IPv4 socket, the addresses returned will be all IPv4 addresses. If sd is an IPv6 socket, the addresses returned can be a mix of IPv4 or IPv6 addresses.

For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored.

If the id field is set to the value '0' then the locally bound addresses are returned without regard to any particular association.

On success, sctp_getladdrs() returns the number of local addresses bound to the socket. If the socket is unbound, sctp_getladdrs() returns 0, and the value of *addrs is undefined. If an error occurs, sctp_getladdrs() returns -1, and the value of *addrs is undefined.

8.6 sctp_freeladdrs()

sctp_freeladdrs() frees all resources allocated by sctp_getladdrs(). Its syntax is,

```
void sctp_freeladdrs(struct sockaddr *addrs);
```

addrs is the array of peer addresses returned by sctp_getladdrs().

8.7 sctp_sendmsg()

An implementation may provide a library function (or possibly system call) to assist the user with the advanced features of SCTP.

sctp_sendmsg(). Its syntax is,

```
ssize_t sctp_sendmsg(int sd,
                  const void *msg,
                  size_t len,
                  const struct sockaddr *to,
                  socklen_t tolen,
                  uint32_t ppid,
                  uint32_t flags,
                  uint16_t stream_no,
                  uint32_t timetolive,
                  uint32_t context)
```

```
sd         - is the socket descriptor
msg        - is the message to be sent.
len        - is the length of the message.
to         - is the destination address of the message.
tolen      - is the length of the destination address.
ppid       - is the same as sinfo_ppid (see section 5.2.2)
flags      - is the same as sinfo_flags (see section 5.2.2)
stream_no  - is the same as sinfo_stream (see section 5.2.2)
timetolive - is the same as sinfo_timetolive (see section 5.2.2)
context    - is the same as sinfo_context (see section 5.2.2)
```

8.8 sctp_recvmsg()

An implementation may provide a library function (or possibly system call) to assist the user with the advanced features of SCTP. Note that in order for the sctp_sndrcvinfo structure to be filled in by sctp_recvmsg() the caller must enable the sctp_data_io_events with the SCTP_EVENTS option.

sctp_recvmsg(). Its syntax is,

```
ssize_t sctp_recvmsg(int sd,
                  void *msg,
                  size_t len,
                  struct sockaddr *from,
                  socklen_t *fromlen
                  struct sctp_sndrcvinfo *sinfo
                  int *msg_flags)

sd         - is the socket descriptor
msg        - is a message buffer to be filled.
len        - is the length of the message buffer.
from       - is a pointer to a address to be filled with
             the sender of this messages address.
fromlen    - is the from length.
sinfo      - A pointer to a sctp_sndrcvinfo structure
             to be filled upon receipt of the message.
msg_flags  - A pointer to a integer to be filled with
             any message flags (e.g. MSG_NOTIFICATION).
```

8.9 sctp_connectx()

An implementation may provide a library function (or possibly system call) to assist the user with associating to an endpoint that is multi-homed. Much like sctp_bindx() this call allows a caller to specify multiple addresses at which a peer can be reached. The way the SCTP stack uses the list of addresses to set up the association is implementation dependant. This function only specifies that the stack will try to make use of all the addresses in the list when needed.

Note that the list of addresses passed in is only used for setting up the association. It does not necessarily equal the set of addresses the peer uses for the resulting association. If the caller wants to find out the set of peer addresses, it must use sctp_getpaddrs() to retrieve them after the association has been set up.

sctp_connectx(). Its syntax is,

```
int sctp_connectx(int sd,
                  struct sockaddr *addrs,
                  int addrcnt)

sd         - is the socket descriptor
addrs      - is an array of addresses.
```

addrcnt - is the number of addresses in the array.

8.10 sctp_send()

An implementation may provide another alternative function or system call to assist an application with the sending of data without the use of the CMSG header structures. The function takes the following form:

sctp_send(). Its syntax is,

```
int sctp_send(int sd,
              const void *msg,
              size_t len,
              const struct sctp_sndrcvinfo *sinfo,
              int flags);
```

sd - is the socket descriptor
msg - The message to be sent
len - The length of the message
sinfo - A pointer to a sctp_sndrcvinfo struture used as described in 5.2.2 for a sendmsg call.
flags - is used in the same format as the sendmsg call flags (e.g. MSG_DONTROUTE).

This function call may also be used to terminate an association using an association identification by setting the sinfo.sinfo_flags to MSG_EOF and the sinfo.sinf_associd to the association that needs to be terminated. In such a case the len of the message would be zero.

8.11 sctp_sendx()

An implementation may provide another alternative function or system call to assist an application with the sending of data without the use of the CMSG header structures that also gives a list of addresses. The list of addresses is provided for implicit association setup. In such a case the list of addresses serves the same purpose as the addresses given in sctp_connectx (see Section 8.9).

sctp_sendx(). Its syntax is,

```
int sctp_sendx(int sd,
               const void *msg,
               size_t len,
            struct sockaddr *addrs,
               int addrcnt,
               struct sctp_sndrcvinfo *sinfo,
               int flags);
```

```
sd         - is the socket descriptor
msg        - The message to be sent
len        - The length of the message
addrs      - is an array of addresses.
addrcnt    - is the number of addresses in the array.
sinfo      - A pointer to a sctp_sndrcvinfo struture used
             as described in 5.2.2 for a sendmsg call.
flags      - is used in the same format as the sendmsg call
             flags (e.g. MSG_DONTROUTE).
```

Note that on return from this call the sinfo structure will have changed in that the sinfo_assoc_id will be filled in with the new association id.

This function call may also be used to terminate an association using an association identification by setting the sinfo.sinfo_flags to MSG_EOF and the sinfo.sinf_associd to the association that needs to be terminated. In such a case the len of the message would be zero.

9. Preprocessor Constants

For application portability it is desireable to define pre-processor constants for determination if sctp is present and supports various features. The following pre-processor constants should be defined in a include file, sctp.h.

HAVE_SCTP - If this constant is defined to 1, then an implementation of SCTP is available.

HAVE_KERNEL_SCTP - If this constant is defined to 1, then a kernel SCTP implementation is available through the sockets interface.

HAVE_SCTP_PRSCTP - If this constant is defined to 1, then the SCTP implementation supports the partial reliablility extension to SCTP.

HAVE_SCTP_ADDIP - If this constant is defined to 1, then the SCTP implementation supports the dynamic address extension to SCTP.

HAVE_SCTP_CANSET_PRIMARY - If this constant is defined to 1, then the SCTP implementation supports the ability to request setting of the remote primary address.

HAVE_SCTP_SAT_NETWORK_CAPABILITY - If this constant is defined to 1, then the SCTP implementation supports the satellite network extension to SCTP.

HAVE_SCTP_MULTIBUF - If this constant is defined to 1, then the SCTP implementation dedicates separate buffer space to each association on a one-to-many socket. If this constant is defined to 0, then the implementation provides a single block of shared buffer space for a one-to-many socket.

HAVE_SCTP_NOCONNECT - If this constant is defined to 1, then the SCTP implementation supports initiating an association on a one-to-one style socket without the use of connect(), as outlined in Section 4.1.5.

10. Security Considerations

Many TCP and UDP implementations reserve port numbers below 1024 for privileged users. If the target platform supports privileged users, the SCTP implementation SHOULD restrict the ability to call bind() or sctp_bindx() on these port numbers to privileged users.

Similarly unpriviledged users should not be able to set protocol parameters which could result in the congestion control algorithm being more aggressive than permitted on the public Internet. These parameters are:

struct sctp_rtoinfo

If an unprivileged user inherits a one-to-many style socket with open associations on a privileged port, it MAY be permitted to accept new associations, but it SHOULD NOT be permitted to open new associations. This could be relevant for the r* family of protocols.

11. Acknowledgments

Special acknowledgment is givne to Ken Fujita who helped extensively in the early formation of this document.

The authors also wish to thank Kavitha Baratakke, Mike Bartlett, Jon Berger, Scott Kimble, Renee Revis, and many others on the TSVWG mailing list for contributing valuable comments.

A special thanks to Phillip Conrad, for his suggested text, quick and constructive insights, and most of all his persistent fighting to keep the interface to SCTP usable for the application programmer.

12. References

[1] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[2] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[3] Braden, B., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.

[4] Bradner, S., "The Internet Standards Process -- Revision 3", BCP 9, RFC 2026, October 1996.

[5] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[6] Stevens, W. and M. Thomas, "Advanced Sockets API for IPv6", RFC 2292, February 1998.

[7] Gilligan, R., Thomson, S., Bound, J. and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 2553, March 1999.

[8] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L. and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

Authors' Addresses

Randall R. Stewart
Cisco Systems, Inc.
4875 Forest Drive
Suite 200
Columbia, SC 29206
USA

Phone:
Email: rrs@cisco.com

Qiaobing Xie
Motorola, Inc.
1501 W. Shure Drive, #2309
Arlington Heights, IL 60004
USA

Phone:
Email: qxie1@email.mot.com

La Monte H.P. Yarroll
TimeSys Corp
925 Liberty Ave.
Pittsburgh, PA 15222
USA

Phone:
Email: piggy@acm.org

Jonathan Wood
DoCoMo USA Labs
181 Metro Drive, Suite 300
San Jose, CA 95110
USA

Phone:
Email: jonwood@speakeasy.net

Kacheong Poon
Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
USA

Phone:
Email: kacheong.poon@sun.com

Michael Tuexen
Univ. of Applied Sciences Muenster
Stegerwaldstr. 39
48565 Steinfurt
Germany

Email: tuexen@fh-muenster.de

Appendix A. one-to-one style Code Example

The following code is a simple implementation of an echo server over SCTP. The example shows how to use some features of one-to-one style IPv4 SCTP sockets, including:

- o Opening, binding, and listening for new associations on a socket;
- o Enabling ancillary data
- o Enabling notifications
- o Using ancillary data with sendmsg() and recvmsg()
- o Using MSG_EOR to determine if an entire message has been read
- o Handling notifications

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <stdlib.h>
#include <unistd.h>
#include <netinet/sctp.h>
#include <sys/uio.h>

#define BUFLEN  100

static void
handle_event(void *buf)
{
        struct sctp_assoc_change *sac;
        struct sctp_send_failed *ssf;
        struct sctp_paddr_change *spc;
        struct sctp_remote_error *sre;
        union sctp_notification *snp;
        char addrbuf[INET6_ADDRSTRLEN];
        const char *ap;
        struct sockaddr_in *sin;
        struct sockaddr_in6 *sin6;

        snp = buf;

        switch (snp->sn_header.sn_type) {
        case SCTP_ASSOC_CHANGE:
                sac = &snp->sn_assoc_change;
                printf("^^^ assoc_change: state=%hu, error=%hu, instr=%hu "
                    "outstr=%hu\n", sac->sac_state, sac->sac_error,
                    sac->sac_inbound_streams, sac->sac_outbound_streams);
                break;
        case SCTP_SEND_FAILED:
```

```
                ssf = &snp->sn_send_failed;
                printf("^^^ sendfailed: len=%hu err=%d\n", ssf->ssf_length,
                    ssf->ssf_error);
                break;

        case SCTP_PEER_ADDR_CHANGE:
                spc = &snp->sn_paddr_change;
                if (spc->spc_aaddr.ss_family == AF_INET) {
                  sin = (struct sockaddr_in *)&spc->spc_aaddr;
                  ap = inet_ntop(AF_INET, &sin->sin_addr,
                                 addrbuf, INET6_ADDRSTRLEN);
                } else {
                  sin6 = (struct sockaddr_in6 *)&spc->spc_aaddr;
                  ap = inet_ntop(AF_INET6, &sin6->sin6_addr,
                                 addrbuf, INET6_ADDRSTRLEN);
                }
                printf("^^^ intf_change: %s state=%d, error=%d\n", ap,
                       spc->spc_state, spc->spc_error);
                break;
        case SCTP_REMOTE_ERROR:
                sre = &snp->sn_remote_error;
                printf("^^^ remote_error: err=%hu len=%hu\n",
                    ntohs(sre->sre_error), ntohs(sre->sre_length));
                break;
        case SCTP_SHUTDOWN_EVENT:
                printf("^^^ shutdown event\n");
                break;
        default:
                printf("unknown type: %hu\n", snp->sn_header.sn_type);
                break;
        }
}

static void *
mysctp_recvmsg(int fd, struct msghdr *msg, void *buf, size_t *buflen,
    ssize_t *nrp, size_t cmsglen)
{
        ssize_t nr = 0, nnr = 0;
        struct iovec iov[1];

        *nrp = 0;
        iov->iov_base = buf;
        iov->iov_len = *buflen;
        msg->msg_iov = iov;
        msg->msg_iovlen = 1;

        for (;;) {
```

```
   #ifndef MSG_XPG4_2
   #define MSG_XPG4_2 0
   #endif
                   msg->msg_flags = MSG_XPG4_2;
                   msg->msg_controllen = cmsglen;

                   nnr = recvmsg(fd, msg, 0);
                   if (nnr <= 0) {
                           /* EOF or error */
                           *nrp = nr;
                           return (NULL);
                   }
                   nr += nnr;

                   if ((msg->msg_flags & MSG_EOR) != 0) {
                           *nrp = nr;
                           return (buf);
                   }

                   /* Realloc the buffer? */
                   if (*buflen == (size_t)nr) {
                           buf = realloc(buf, *buflen * 2);
                           if (buf == 0) {
                                   fprintf(stderr, "out of memory\n");
                                   exit(1);
                           }
                           *buflen *= 2;
                   }

                   /* Set the next read offset */
                   iov->iov_base = (char *)buf + nr;
                   iov->iov_len = *buflen - nr;

           }
   }

   static void
   echo(int fd, int socketModeone_to_many)
   {
           ssize_t nr;
           struct sctp_sndrcvinfo *sri;
           struct msghdr msg[1];
           struct cmsghdr *cmsg;
           char cbuf[sizeof (*cmsg) + sizeof (*sri)];
           char *buf;
           size_t buflen;
           struct iovec iov[1];
           size_t cmsglen = sizeof (*cmsg) + sizeof (*sri);
```

```
        /* Allocate the initial data buffer */
        buflen = BUFLEN;
        if (!(buf = malloc(BUFLEN))) {
                fprintf(stderr, "out of memory\n");
                exit(1);
        }

        /* Set up the msghdr structure for receiving */
        memset(msg, 0, sizeof (*msg));
        msg->msg_control = cbuf;
        msg->msg_controllen = cmsglen;
        msg->msg_flags = 0;
        cmsg = (struct cmsghdr *)cbuf;
        sri = (struct sctp_sndrcvinfo *)(cmsg + 1);

        /* Wait for something to echo */
        while (buf = mysctp_recvmsg(fd, msg, buf, &buflen, &nr, cmsglen)) {

                /* Intercept notifications here */
                if (msg->msg_flags & MSG_NOTIFICATION) {
                        handle_event(buf);
                        continue;
                }

                iov->iov_base = buf;
                iov->iov_len = nr;
                msg->msg_iov = iov;
                msg->msg_iovlen = 1;

                printf("got %u bytes on stream %hu:\n", nr,
                    sri->sinfo_stream);
                write(0, buf, nr);

                /* Echo it back */
                msg->msg_flags = MSG_XPG4_2;
                if (sendmsg(fd, msg, 0) < 0) {
                        perror("sendmsg");
                        exit(1);
                }
        }

        if (nr < 0) {
                perror("recvmsg");
        }
        if(socketModeone_to_many == 0)
          close(fd);
}
```

```
   int main()
   {
           struct sctp_event_subscribe event;
           int lfd, cfd;
           int onoff = 1;
           struct sockaddr_in sin[1];

           if ((lfd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) == -1) {
                   perror("socket");
                   exit(1);
           }

           sin->sin_family = AF_INET;
           sin->sin_port = htons(7);
           sin->sin_addr.s_addr = INADDR_ANY;
           if (bind(lfd, (struct sockaddr *)sin, sizeof (*sin)) == -1) {
                   perror("bind");
                   exit(1);
           }

           if (listen(lfd, 1) == -1) {
                   perror("listen");
                   exit(1);
           }

           /* Wait for new associations */
           for (;;) {
                   if ((cfd = accept(lfd, NULL, 0)) == -1) {
                           perror("accept");
                           exit(1);
                   }

                   /* Enable all events */
                   event.sctp_data_io_event = 1;
                   event.sctp_association_event = 1;
                   event.sctp_address_event = 1;
                   event.sctp_send_failure_event = 1;
                   event.sctp_peer_error_event = 1;
                   event.sctp_shutdown_event = 1;
                   event.sctp_partial_delivery_event = 1;
                   event.sctp_adaption_layer_event = 1;
                   if (setsockopt(cfd, IPPROTO_SCTP,
                       SCTP_EVENTS, &event,
                       sizeof(event)) != 0) {
                       perror("setevent failed");
                       exit(1);
                   }
                   /* Echo back any and all data */
```

```
                 echo(cfd,0);
         }
 }
```

Appendix B. one-to-many style Code Example

The following code is a simple implementation of an echo server over SCTP. The example shows how to use some features of one-to-many style IPv4 SCTP sockets, including:

- o Opening and binding of a socket;
- o Enabling ancillary data
- o Enabling notifications
- o Using ancillary data with sendmsg() and recvmsg()
- o Using MSG_EOR to determine if an entire message has been read
- o Handling notifications

Note most functions defined in Appendix A are reused in thi
s example.

```
int main()
{
        int fd;
        int idleTime = 2;
        struct sockaddr_in sin[1];
        struct sctp_event_subscribe event;

        if ((fd = socket(AF_INET, SOCK_SEQPACKET, IPPROTO_SCTP)) == -1) {
          perror("socket");
          exit(1);
        }

        sin->sin_family = AF_INET;
        sin->sin_port = htons(7);
        sin->sin_addr.s_addr = INADDR_ANY;
        if (bind(fd, (struct sockaddr *)sin, sizeof (*sin)) == -1) {
          perror("bind");
          exit(1);
        }

        /* Enable all notifications and events */
        event.sctp_data_io_event = 1;
        event.sctp_association_event = 1;
        event.sctp_address_event = 1;
        event.sctp_send_failure_event = 1;
        event.sctp_peer_error_event = 1;
        event.sctp_shutdown_event = 1;
        event.sctp_partial_delivery_event = 1;
        event.sctp_adaption_layer_event = 1;
        if (setsockopt(fd, IPPROTO_SCTP,
            SCTP_EVENTS, &event,
            sizeof(event)) != 0) {
```

```
                perror("setevent failed");
                exit(1);
            }
            /* Set associations to auto-close in 2 seconds of
             * inactivity
             */
            if (setsockopt(fd, IPPROTO_SCTP, SCTP_AUTOCLOSE,
                           &idleTime, 4) < 0) {
              perror("setsockopt SCTP_AUTOCLOSE");
              exit(1);
            }

            /* Allow new associations to be accepted */
            if (listen(fd, 1) < 0) {
              perror("listen");
              exit(1);
            }

            /* Wait for new associations */
            while(1){
              /* Echo back any and all data */
              echo(fd,1);
            }
   }
```

Acknowledgment

Funding for the RFC Editor function is currently provided by the Internet Society.