Internet Engineering Task Force (IETF)
Request for Comments: 6458
Category: Informational
ISSN: 2070-1721

R. Stewart
Adara Networks
M. Tuexen
Muenster Univ. of Appl. Sciences
K. Poon
Oracle Corporation
P. Lei
Cisco Systems, Inc.
V. Yasevich
HP

December 2011

Sockets API Extensions for the Stream Control Transmission Protocol (SCTP)

Abstract

This document describes a mapping of the Stream Control Transmission Protocol (SCTP) 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.

Status of This Memo

This document is not an Internet Standards Track specification; it is published for informational purposes.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Not all documents approved by the IESG are a candidate for any level of Internet Standard; see Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc6458.

Table of Contents

1. Introduction

The sockets API has provided a standard mapping of the Internet Protocol suite to many operating systems. Both TCP [RFC0793] and UDP [RFC0768] 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 connectionless protocols, such as UDP. A one-to-many style SCTP socket should be able to control multiple SCTP associations. This is similar to a UDP socket, which can communicate with many peer endpoints. Each of these associations is assigned an association identifier so that an

application 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 to be ported to use SCTP with very little effort. Developers familiar with these semantics can easily adapt to SCTP. Another purpose is to make sure that existing mechanisms in most operating systems that support sockets, 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 this document defines 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 the applications.

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

Some of the SCTP mechanisms cannot be adequately mapped to an existing socket interface. In some cases, it is more desirable to have a new interface instead of using existing socket calls. Section 9 of this document describes these new interfaces.

Please note that some elements of the SCTP sockets API are declared as deprecated. During the evolution of this document, elements of the API were introduced, implemented, and later on replaced by other elements. These replaced elements are declared as deprecated, since they are still available in some implementations and the replacement functions are not. This applies especially to older versions of operating systems supporting SCTP. New SCTP socket implementations must implement at least the non-deprecated elements. Implementations intending interoperability with older versions of the API should also include the deprecated functions.

2. Data Types

Whenever possible, Portable Operating System Interface (POSIX) data types defined in [IEEE-1003.1-2008] are used: uintN_t means an unsigned integer of exactly N bits (e.g., uint16_t). This document also assumes the argument data types from POSIX when possible (e.g., the final argument to setsockopt() is a socklen_t value). Whenever buffer sizes are specified, the POSIX size_t data type is used.

3. One-to-Many Style Interface

In the one-to-many style interface, there is a one-to-many relationship between sockets and associations.

3.1. Basic Operation

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

- socket()
- bind()
- listen()
- recvmsg()
- sendmsg()
- close()

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

- socket()
- sendmsg()
- recvmsg()
- close()

In this style, by default, all of the associations connected to the endpoint are represented with a single socket. Each association is assigned an association identifier (the type is sctp_assoc_t) so that an application can use it to differentiate among them. In some implementations, the peer endpoints' 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 set up an association implicitly. It cannot be used to send data to an established association, as the association identifier cannot be specified.

Once an association identifier is assigned to an SCTP association, that identifier will not be reused until the application explicitly terminates the use of the association. The resources belonging 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 8.1.8) is set. In this case, after the association is terminated gracefully and automatically, the association identifier 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 endpoint.

If the server or client wishes to branch an existing association off to a separate socket, it is required to call sctp_peeloff() and to specify the association identifier. The sctp_peeloff() call will return a new one-to-one style socket that can then be used with recv() and send() functions for message passing. See Section 9.2 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.

One-to-many style socket calls are discussed in more detail in the following subsections.

3.1.1. socket()

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

The function prototype is

```
int socket(int domain,
           int type,
           int protocol);
```

and one uses PF_INET or PF_INET6 as the domain, SOCK_SEQPACKET as the type, and IPPROTO_SCTP as the protocol.

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

The function returns a socket descriptor, or -1 in case of an error.

Using the PF_INET domain indicates the creation of an endpoint that can use only IPv4 addresses, while PF_INET6 creates an endpoint that can use both IPv6 and IPv4 addresses.

3.1.2. bind()

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

An SCTP endpoint can be associated with multiple addresses. To do this, sctp_bindx() is introduced in Section 9.1 to help applications do the job of associating multiple addresses. But note that an endpoint can only be associated with one local port.

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 [RFC4960].

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

The function prototype of bind() is

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

and the arguments are

sd: The socket descriptor returned by socket().

addr: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3493]).

addrlen: The size of the address structure.

bind() returns 0 on success and -1 in case of an error.

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 the IP address part of 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 the IPv4 sin_port or IPv6 sin6_port is set to 0, the operating system will choose an ephemeral port for the endpoint.

If 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.

The completion of this bind() process does not allow 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.

3.1.3. listen()

By default, a one-to-many style socket does not accept new association requests. An application uses listen() to mark a socket as being able to accept new associations.

The function prototype is

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

and the arguments are

sd: The socket descriptor of the endpoint.

backlog: If backlog is non-zero, enable listening, else disable listening.

listen() returns 0 on success and -1 in case of an error.

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 only actively initiated associations are possible on the socket. 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 identifier for a new association, so if applications wish to use the association identifier as a parameter to other socket calls, they should ensure that the SCTP_ASSOC_CHANGE event is enabled.

3.1.4. sendmsg() and recvmsg()

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

The function prototypes are

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

and

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

using the following arguments:

sd: The socket descriptor of the endpoint.

message: Pointer to the msghdr structure that contains a single user message and possibly some ancillary data. See Section 5 for a 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.

sendmsg() returns the number of bytes accepted by the kernel or -1 in case of an error. recvmsg() returns the number of bytes received or -1 in case of an error.

As described in Section 5, different types of ancillary data can be sent and received along with user data. When sending, the ancillary data is used to specify the sent behavior, such as the SCTP stream number to use. When receiving, the ancillary data is used to describe the received data, such as the SCTP stream sequence number of the message.

When sending user data with sendmsg(), the msg_name field in the msghdr structure will be filled with one of the transport addresses of the intended receiver. If there is no existing association 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 7.5 for more on implicit association setup). If sendmsg() is called with no data and there is no existing association, a new one will be established. The SCTP_INIT type ancillary data can be used to change some of the parameters used to set up a new association. If sendmsg() is called with NULL data, and there is no existing association but the SCTP_ABORT or SCTP_EOF flags are set as described in Section 5.3.4, then -1 is returned and errno is set to EINVAL. Sending a message using sendmsg() is atomic unless explicit end of record (EOR) marking is enabled on the socket specified by sd (see Section 8.1.26).

If a peer sends a SHUTDOWN, an 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 the 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 identifier.

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 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 in any stream. The socket option SCTP_FRAGMENT_INTERLEAVE controls various aspects of what interlacing of messages occurs for both the one-to-one and the one-to-many style sockets. Please consult Section 8.1.20 for further details on message delivery options.

3.1.5. close()

Applications use close() to perform graceful shutdown (as described in Section 10.1 of [RFC4960]) on all of the associations currently represented by a one-to-many style socket.

The function prototype is

```
int close(int sd);
```

and the argument is

sd: The socket descriptor of the associations to be closed.

0 is returned on success and -1 in case of an error.

To gracefully shut down a specific association represented by the one-to-many style socket, an application should use the sendmsg() call and include the SCTP_EOF flag. A user may optionally terminate an association non-gracefully by using sendmsg() with the SCTP_ABORT flag set and possibly passing a user-specified abort code in the data field. Both flags SCTP_EOF and SCTP_ABORT are passed with ancillary data (see Section 5.3.4) 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()

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

The function prototype is

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

and the arguments are

sd: The socket descriptor to which a new association is added.

nam: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3493]).

len: The size of the address.

0 is returned on success and -1 in case of an error.

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.

Note that SCTP allows data exchange, similar to T/TCP [RFC1644] (made Historic by [RFC6247]), during the association setup phase. If an application wants to do this, it cannot use the connect() call. Instead, it should use sendto() or sendmsg() to initiate an association. If it uses sendto() and it wants to change the initialization behavior, it needs to use the SCTP_INITMSG socket option before calling sendto(). Or it can use sendmsg() with SCTP_INIT type ancillary data to initiate an association without calling setsockopt(). Note that the implicit setup is supported for the one-to-many style sockets.

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 an SCTP socket.

3.2. Non-Blocking Mode

Some SCTP applications may wish to avoid being blocked when calling a socket interface function.

Once a bind() call and/or subsequent sctp_bindx() calls are complete on a one-to-many style socket, an application may set the non-blocking option via a fcntl() (such as O_NONBLOCK). After setting the socket to non-blocking mode, the sendmsg() function returns immediately. The success or failure of sending the data message (with possible SCTP_INITMSG ancillary data) will be signaled by the SCTP_ASSOC_CHANGE event with SCTP_COMM_UP or SCTP_CANT_START_ASSOC. If user data could not be sent (due to an SCTP_CANT_START_ASSOC), the sender will also receive an SCTP_SEND_FAILED_EVENT event. Events can be received by the user calling recvmsg(). A server (having called listen()) is also

notified of an association-up event via the reception of an SCTP_ASSOC_CHANGE with SCTP_COMM_UP via the calling of recvmsg() and possibly the reception of the first data message.

To shut down the association gracefully, the user must call sendmsg() with no data and with the SCTP_EOF flag set as described in Section 5.3.4. The function returns immediately, and completion of the graceful shutdown is indicated by an SCTP_ASSOC_CHANGE notification of type SCTP_SHUTDOWN_COMP (see Section 6.1.1). Note that this can also be done using the sctp_sendv() call described in Section 9.12.

It is recommended that an application use caution when using select() (or poll()) for writing on a one-to-many style socket, because the 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 style socket is writable. An application that writes after the select() returns may still block, since the association that was writable is not the destination association of the write call. Likewise, select() (or poll()) for reading from a one-to-many style 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_peeloff() function (see Section 9.2) to separate the association of interest from the one-to-many style socket.

Note that some implementations may have an extended select call, such as epoll or kqueue, that may escape this limitation and allow a select on a specific association of a one-to-many style socket, but this is an implementation-specific detail that a portable application cannot depend on.

3.3. 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 implementers 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 the send socket buffer is completely filled. This results in 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, there are multiple associations for a single socket, which 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 implementer 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 implementer 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

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

- o 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

- o 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).

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.

One-to-one style sockets can be connected (explicitly or implicitly) at most once, similar to TCP sockets.

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.

4.1. Basic Operation

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

- o socket()

- o bind()

- o listen()

- o 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

- o close()

to terminate the association.

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

- socket()
- connect()

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

The client calls

- close()

to terminate this association when done.

4.1.1. socket()

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

The function prototype is

```
int socket(int domain,
           int type,
           int protocol);
```

and one uses PF_INET or PF_INET6 as the domain, SOCK_STREAM as the type, and IPPROTO_SCTP as the protocol.

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

Using the PF_INET domain indicates the creation of an endpoint that can use only IPv4 addresses, while PF_INET6 creates an endpoint that can use both IPv6 and IPv4 addresses.

4.1.2. bind()

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

An SCTP endpoint can be associated with multiple addresses. To do this, sctp_bindx() is introduced in Section 9.1 to help applications do the job of associating multiple addresses. But note that an endpoint can only be associated with one local port.

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 [RFC4960].

The function prototype of bind() is

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

and the arguments are

sd: The socket descriptor returned by socket().

addr: The address structure (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3493]).

addrlen: The size of the address structure.

If sd is an IPv4 socket, the address passed must be an IPv4 address. If 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 the IP address part of 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 the IPv4 sin_port or IPv6 sin6_port is set to 0, the operating system will choose an ephemeral port for the endpoint.

If 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 these 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 allow 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()

Applications use listen() to allow the SCTP endpoint to accept inbound associations.

The function prototype is

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

and the arguments are

sd: The socket descriptor of the SCTP endpoint.

backlog: 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 [RFC4960]) and are in the ESTABLISHED state. Note that a backlog of '0' indicates that the caller no longer wishes to receive new associations.

listen() returns 0 on success and -1 in case of an error.

4.1.4. accept()

Applications use the 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 function prototype is

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

and the arguments are

sd: The listening socket descriptor.

addr: On return, addr (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3493]) will contain the primary address of the peer endpoint.

addrlen: On return, addrlen will contain the size of addr.

The function returns the socket descriptor for the newly formed association on success and -1 in case of an error.

4.1.5. connect()

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

The function prototype is

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

and the arguments are

sd: The socket descriptor of the endpoint.

addr: The peer's (struct sockaddr_in for an IPv4 address or struct sockaddr_in6 for an IPv6 address; see [RFC3493]) address.

addrlen: The size of the address.

connect() returns 0 on success and -1 on error.

This operation corresponds to the ASSOCIATE primitive described in Section 10.1 of [RFC4960].

The number of outbound streams the new association has is stack dependent. Before connecting, applications can use the SCTP_INITMSG option described in Section 8.1.3 to change the number of outbound streams.

If 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_INIT for IPv4 and IPv6 sockets, respectively. One of the 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] (made Historic by [RFC6247]), during the association setup phase. If an application wants to do this, it cannot use the connect() call. Instead, it should use sendto() or sendmsg() to initiate an association. If it uses sendto() and it wants to change the initialization behavior, it needs to use the SCTP_INITMSG socket option before calling sendto(). Or it can use sendmsg() with SCTP_INIT type ancillary data to initiate an association without calling setsockopt(). Note that the implicit setup is supported for the one-to-one style sockets.

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 an SCTP socket.

4.1.6. close()

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

The function prototype is

```
int close(int sd);
```

and the argument is

sd: The socket descriptor of the association to be closed.

close() returns 0 on success and -1 in case of an error.

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

4.1.7. shutdown()

SCTP differs from TCP in that it does not have half close 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 function prototype is

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

and the arguments are

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.

shutdown() returns 0 on success and -1 in case of an error.

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 close state. SHUT_RD behaves the same for SCTP as for TCP. The purpose of SCTP SHUT_WR is to close the SCTP association while still leaving the socket descriptor open. This allows the caller to receive back any data that SCTP is unable to deliver (see Section 6.1.4 for more information) and receive event notifications.

To perform the ABORT operation described in Section 10.1 of [RFC4960], an application can use the socket option SO_LINGER. SO_LINGER is described in Section 8.1.4.

4.1.8. sendmsg() and recvmsg()

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 (see Section 3.1.4), 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 socket is connected and the transport address given is not part of the current association, the data will not be sent, and an SCTP_SEND_FAILED_EVENT event will be delivered to the application if send failure events are enabled.

2. Using sendmsg() on a non-connected one-to-one style socket for implicit connection setup may or may not work, depending on the SCTP implementation.

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 may not work with one-to-many style sockets, depending on the implementation. See Section 9.3 for a multi-homed style version of the call.

The function prototype is

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

and the arguments are

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.

getpeername() returns 0 on success and -1 in case of an error.

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

This section discusses important data structures that 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 detail in [RFC3542]. They are defined as

```
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 */
  int 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 */
};

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 Sections 3.1.4 and 4.1.8).

The scatter/gather buffers, or I/O vectors (pointed to by the msg_iov field) are treated by SCTP as a single user message for both sendmsg() and recvmsg().

The SCTP stack uses the ancillary data (msg_control field) to communicate the attributes, such as SCTP_RCVINFO, of the message stored in msg_iov to the socket endpoint. The different ancillary data types are described in Section 5.3.

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

If a notification has arrived, recvmsg() will return the notification in the msg_iov field and set the MSG_NOTIFICATION flag in msg_flags. If the MSG_NOTIFICATION flag is not set, recvmsg() will return data. See Section 6 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. Ancillary Data Considerations and Semantics

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

5.2.1. Multiple Items and Ordering

Multiple ancillary data items may be included in any call to sendmsg() or recvmsg(); these may include multiple SCTP items, non-SCTP items (such as IP-level 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/SCTP_SNDINFO/SCTP_RCVINFO type ancillary data always corresponds to the data in the msghdr's msg_iov member. There can be only one such type of ancillary data for each sendmsg() or recvmsg() call.

5.2.2. 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 [RFC3542] and the SCTP implementation's documentation for more information. The following is an example, from [RFC3542], 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_len == 0) {
     /* Error handling */
     break;
  }
  if (cmsgptr->cmsg_level == ... && cmsgptr->cmsg_type == ... ) {
    u_char  *ptr;

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

5.2.3. Control Message Buffer Sizing

The information conveyed via SCTP_SNDRCV/SCTP_SNDINFO/SCTP_RCVINFO ancillary data will often be fundamental to the correct and sane operation of the sockets application. This is particularly true for one-to-many style sockets, but also for one-to-one style sockets. For example, if an application needs to send and receive data on different SCTP streams, SCTP_SNDRCV/SCTP_SNDINFO/SCTP_RCVINFO ancillary data is 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 of 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 [RFC3542], 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 (buf) for msg_control on each call to recvmsg() and be assured that we would not lose any ancillary data to truncation.

5.3. 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 msghdr structure'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 preceded by a struct cmsghdr (see Section 5.1), which defines the function and purpose of the data contained in the cmsg_data[] member.

By default, on either style of socket, SCTP will pass no ancillary data. Specific ancillary data items can be enabled with socket options defined for SCTP; see Section 6.2.

Note that all ancillary types are of fixed length; see Section 5.2 for further discussion on this. These data structures use struct sockaddr_storage (defined in [RFC3493]) 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 sockets API definitions ([RFC3542] and [RFC3493]) define a number of ancillary data items. If a sockets 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 those 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.

5.3.1. 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

The sctp_initmsg structure is defined below:

```
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: This is an integer representing the number of streams to which the application wishes to be able to send. 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 the use of the endpoint's default value.

sinit_max_instreams: 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 the use of the endpoint's default value.

sinit_max_attempts: 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 the use of the endpoint's default value. This is normally set to the system's default 'Max.Init.Retransmit' value.

sinit_max_init_timeo: This value represents the largest timeout or retransmission timeout (RTO) value (in milliseconds) to use in attempting an 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 the use of the endpoint's default value. This is normally set to the system's 'RTO.Max' value (60 seconds).

5.3.2. SCTP Header Information Structure (SCTP_SNDRCV) - DEPRECATED

This cmsghdr structure specifies SCTP options for sendmsg() and describes SCTP header information about a received message through recvmsg(). This structure mixes the send and receive path. SCTP_SNDINFO (described in Section 5.3.4) and SCTP_RCVINFO (described in Section 5.3.5) split this information. These structures should be used, when possible, since SCTP_SNDRCV is deprecated.

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_SNDRCV	struct sctp_sndrcvinfo

The sctp_sndrcvinfo structure is defined below:

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

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

recvmsg() flags:

SCTP_UNORDERED: This flag is present when the message was sent unordered.

sendmsg() flags:

SCTP_UNORDERED: This flag requests the unordered delivery of the message. If this flag is clear, the datagram is considered an ordered send.

SCTP_ADDR_OVER: This flag, for a one-to-many style socket, requests that the SCTP stack override the primary destination address with the address found with the sendto/ sendmsg call.

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

SCTP_EOF: Setting this flag invokes the SCTP graceful shutdown procedure on the specified association. Graceful shutdown assures that all data queued by both endpoints is successfully transmitted before closing the association.

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

sinfo_ppid: This value in sendmsg() is an unsigned integer 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 the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the htonl() computation.

sinfo_context: 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 an error occurs on the send of a message and is retrieved with each undelivered message.

sinfo_timetolive: For the sending side, this field contains the message's time to live, in milliseconds. The sending side will expire the message within the specified time period if the message has 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: For the receiving side, this field holds a Transmission Sequence Number (TSN) that was assigned to one of the SCTP DATA chunks. For the sending side, it is ignored.

sinfo_cumtsn: This field will hold the current cumulative TSN as known by the underlying SCTP layer. Note that this field is ignored when sending.

sinfo_assoc_id: 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. This field is ignored for one-to-one style sockets.

An sctp_sndrcvinfo item always corresponds to the data in msg_iov.

5.3.3. Extended SCTP Header Information Structure (SCTP_EXTRCV) - DEPRECATED

This cmsghdr structure specifies SCTP options for SCTP header information about a received message via recvmsg(). Note that this structure is an extended version of SCTP_SNDRCV (see Section 5.3.2) and will only be received if the user has set the socket option SCTP_USE_EXT_RCVINFO (see Section 8.1.22) to true in addition to any event subscription needed to receive ancillary data. Note that data in the next message is not valid unless the current message is completely read, i.e., unless the MSG_EOR is set; in other words, if the application has more data to read from the current message, then no next-message information will be available.

SCTP_NXTINFO (described in Section 5.3.6) should be used when possible, since SCTP_EXTRCV is considered deprecated.

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_EXTRCV	struct sctp_extrcvinfo

The sctp_extrcvinfo structure is defined below:

```
struct sctp_extrcvinfo {
  uint16_t sinfo_stream;
  uint16_t sinfo_ssn;
  uint16_t sinfo_flags;
  uint32_t sinfo_ppid;
  uint32_t sinfo_context;
  uint32_t sinfo_pr_value;
  uint32_t sinfo_tsn;
  uint32_t sinfo_cumtsn;
  uint16_t serinfo_next_flags;
  uint16_t serinfo_next_stream;
  uint32_t serinfo_next_aid;
  uint32_t serinfo_next_length;
  uint32_t serinfo_next_ppid;
  sctp_assoc_t sinfo_assoc_id;
};
```

sinfo_*: Please see Section 5.3.2 for details for these fields.

serinfo_next_flags: This bitmask will hold one or more of the following values:

SCTP_NEXT_MSG_AVAIL: This bit, when set to 1, indicates that next-message information is available; i.e., next_stream, next_aid, next_length, and next_ppid fields all have valid values. If this bit is set to 0, then these fields are not valid and should be ignored.

SCTP_NEXT_MSG_ISCOMPLETE: This bit, when set, indicates that the next message is completely in the receive buffer. The next_length field thus contains the entire message size. If this flag is set to 0, then the next_length field only contains part of the message size, since the message is still being received (it is being partially delivered).

SCTP_NEXT_MSG_IS_UNORDERED: This bit, when set, indicates that the next message to be received was sent by the peer as unordered. If this bit is not set (i.e., the bit is 0) the next message to be read is an ordered message in the stream specified.

SCTP_NEXT_MSG_IS_NOTIFICATION: This bit, when set, indicates that the next message to be received is not a message from the peer, but instead is a MSG_NOTIFICATION from the local SCTP stack.

serinfo_next_stream: This value, when valid (see serinfo_next_flags), contains the next stream number that will be received on a subsequent call to one of the receive message functions.

serinfo_next_aid: This value, when valid (see serinfo_next_flags), contains the next association identifier that will be received on a subsequent call to one of the receive message functions.

serinfo_next_length: This value, when valid (see serinfo_next_flags), contains the length of the next message that will be received on a subsequent call to one of the receive message functions. Note that this length may be a partial length, depending on the settings of next_flags.

serinfo_next_ppid: This value, when valid (see serinfo_next_flags), contains the ppid of the next message that will be received on a subsequent call to one of the receive message functions.

5.3.4. SCTP Send Information Structure (SCTP_SNDINFO)

This cmsghdr structure specifies SCTP options for sendmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_SNDINFO	struct sctp_sndinfo

The sctp_sndinfo structure is defined below:

```
struct sctp_sndinfo {
  uint16_t snd_sid;
  uint16_t snd_flags;
  uint32_t snd_ppid;
  uint32_t snd_context;
  sctp_assoc_t snd_assoc_id;
};
```

snd_sid: This value holds the stream number to which the application wishes to send this message. If a sender specifies an invalid stream number, an error indication is returned and the call fails.

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

SCTP_UNORDERED: This flag requests the unordered delivery of the message. If this flag is clear, the datagram is considered an ordered send.

SCTP_ADDR_OVER: This flag, for a one-to-many style socket, requests that the SCTP stack override the primary destination address with the address found with the sendto()/sendmsg call.

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

SCTP_EOF: Setting this flag invokes the SCTP graceful shutdown procedures on the specified association. Graceful shutdown assures that all data queued by both endpoints is successfully transmitted before closing the association.

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

snd_ppid: This value in sendmsg() is an unsigned integer that is passed to the remote end in each user message. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the htonl() computation.

snd_context: 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 an error occurs on the send of a message and is retrieved with each undelivered message.

snd_assoc_id: 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. This field is ignored for one-to-one style sockets.

An sctp_sndinfo item always corresponds to the data in msg_iov.

5.3.5. SCTP Receive Information Structure (SCTP_RCVINFO)

This cmsghdr structure describes SCTP receive information about a received message through recvmsg().

To enable the delivery of this information, an application must use the SCTP_RECVRCVINFO socket option (see Section 8.1.29).

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_RCVINFO	struct sctp_rcvinfo

The sctp_rcvinfo structure is defined below:

```
struct sctp_rcvinfo {
  uint16_t rcv_sid;
  uint16_t rcv_ssn;
  uint16_t rcv_flags;
  uint32_t rcv_ppid;
  uint32_t rcv_tsn;
  uint32_t rcv_cumtsn;
  uint32_t rcv_context;
  sctp_assoc_t rcv_assoc_id;
};
```

rcv_sid: The SCTP stack places the message's stream number in this value.

rcv_ssn: 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).

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

SCTP_UNORDERED: This flag is present when the message was sent unordered.

rcv_ppid: This value is the same information that was passed by the upper layer in the peer application. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the ntohl() computation.

rcv_tsn: This field holds a TSN that was assigned to one of the SCTP DATA chunks.

rcv_cumtsn: This field will hold the current cumulative TSN as known by the underlying SCTP layer.

rcv_context: This value is an opaque 32-bit context datum that was set by the user with the SCTP_CONTEXT socket option. This value is passed back to the upper layer if an error occurs on the send of a message and is retrieved with each undelivered message.

rcv_assoc_id: 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. This field is ignored for one-to-one style sockets.

An sctp_rcvinfo item always corresponds to the data in msg_iov.

5.3.6. SCTP Next Receive Information Structure (SCTP_NXTINFO)

This cmsghdr structure describes SCTP receive information of the next message that will be delivered through recvmsg() if this information is already available when delivering the current message.

To enable the delivery of this information, an application must use the SCTP_RECVNXTINFO socket option (see Section 8.1.30).

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_NXTINFO	struct sctp_nxtinfo

The sctp_nxtinfo structure is defined below:

```
struct sctp_nxtinfo {
  uint16_t nxt_sid;
  uint16_t nxt_flags;
  uint32_t nxt_ppid;
  uint32_t nxt_length;
  sctp_assoc_t nxt_assoc_id;
};
```

nxt_sid: The SCTP stack places the next message's stream number in this value.

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

SCTP_UNORDERED: This flag is present when the next message was sent unordered.

SCTP_COMPLETE: This flag indicates that the entire message has been received and is in the socket buffer. Note that this has special implications with respect to the nxt_length field; see the description for nxt_length below.

SCTP_NOTIFICATION: This flag is present when the next message is not a user message but instead is a notification.

nxt_ppid: This value is the same information that was passed by the upper layer in the peer application for the next message. Please note that the SCTP stack performs no byte order modification of this field. For example, if the DATA chunk has to contain a given value in network byte order, the SCTP user has to perform the ntohl() computation.

nxt_length: This value is the length of the message currently within the socket buffer. This might NOT be the entire length of the message, since a partial delivery may be in progress. Only if the flag SCTP_COMPLETE is set in the nxt_flags field does this field represent the size of the entire next message.

nxt_assoc_id: The association handle field of the next message, nxt_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. This field is ignored for one-to-one style sockets.

5.3.7. SCTP PR-SCTP Information Structure (SCTP_PRINFO)

This cmsghdr structure specifies SCTP options for sendmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_PRINFO	struct sctp_prinfo

The sctp_prinfo structure is defined below:

```
struct sctp_prinfo {
  uint16_t pr_policy;
  uint32_t pr_value;
};
```

pr_policy: This specifies which Partially Reliable SCTP (PR-SCTP) policy is used. Using SCTP_PR_SCTP_NONE results in a reliable transmission. When SCTP_PR_SCTP_TTL is used, the PR-SCTP policy "timed reliability" defined in [RFC3758] is used. In this case, the lifetime is provided in pr_value.

pr_value: The meaning of this field depends on the PR-SCTP policy specified by the pr_policy field. It is ignored when SCTP_PR_SCTP_NONE is specified. In the case of SCTP_PR_SCTP_TTL, the lifetime in milliseconds is specified.

An sctp_prinfo item always corresponds to the data in msg_iov.

5.3.8. SCTP AUTH Information Structure (SCTP_AUTHINFO)

This cmsghdr structure specifies SCTP options for sendmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_AUTHINFO	struct sctp_authinfo

The sctp_authinfo structure is defined below:

```
struct sctp_authinfo {
  uint16_t auth_keynumber;
};
```

auth_keynumber: This specifies the shared key identifier used for sending the user message.

An sctp_authinfo item always corresponds to the data in msg_iov. Please note that the SCTP implementation must not bundle user messages that need to be authenticated using different shared key identifiers.

5.3.9. SCTP Destination IPv4 Address Structure (SCTP_DSTADDRV4)

This cmsghdr structure specifies SCTP options for sendmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_DSTADDRV4	struct in_addr

This ancillary data can be used to provide more than one destination address to sendmsg(). It can be used to implement sctp_sendv() using sendmsg().

5.3.10. SCTP Destination IPv6 Address Structure (SCTP_DSTADDRV6)

This cmsghdr structure specifies SCTP options for sendmsg().

cmsg_level	cmsg_type	cmsg_data[]
IPPROTO_SCTP	SCTP_DSTADDRV6	struct in6_addr

This ancillary data can be used to provide more than one destination address to sendmsg(). It can be used to implement sctp_sendv() using sendmsg().

6. 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 6.2 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 that provide 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.

6.1. SCTP Notification Structure

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

```
union sctp_notification {
  struct sctp_tlv {
    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_adaptation_event sn_adaptation_event;
  struct sctp_pdapi_event sn_pdapi_event;
  struct sctp_authkey_event sn_auth_event;
  struct sctp_sender_dry_event sn_sender_dry_event;
  struct sctp_send_failed_event sn_send_failed_event;
};
```

sn_type: 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 6.1.1 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 an endpoint via a specific transport address). Please see Section 6.1.2 for data structure details.

SCTP_REMOTE_ERROR: The attached error message is an Operation Error message received from the remote peer. It includes the complete TLV sent by the remote endpoint. See Section 6.1.3 for the detailed format.

SCTP_SEND_FAILED_EVENT: The attached datagram could not be sent to the remote endpoint. This structure includes the original SCTP_SNDINFO that was used in sending this message; i.e., this structure uses the sctp_sndinfo per Section 6.1.11.

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

SCTP_ADAPTATION_INDICATION: This notification holds the peer's indicated adaptation layer. Please see Section 6.1.6.

SCTP_PARTIAL_DELIVERY_EVENT: This notification is used to tell a receiver that the partial delivery has been aborted. This may indicate that the association is about to be aborted. Please see Section 6.1.7.

SCTP_AUTHENTICATION_EVENT: This notification is used to tell a receiver that either an error occurred on authentication, or a new key was made active. See Section 6.1.8.

SCTP_SENDER_DRY_EVENT: This notification is used to inform the application that the sender has no more user data queued for transmission or retransmission. See Section 6.1.9.

sn_flags: These are notification-specific flags.

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

6.1.1. SCTP_ASSOC_CHANGE

Communication notifications inform the application 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[];
};
```

sac_type: This field should be set to SCTP_ASSOC_CHANGE.

sac_flags: This field is currently unused.

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

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

SCTP_COMM_UP: A new association is now ready, and data may be exchanged with this peer. When an association has been established successfully, this notification should be the first one.

SCTP_COMM_LOST: The association has failed. The association is now in the closed state. If SEND_FAILED notifications are turned on, an SCTP_COMM_LOST is accompanied by a series of SCTP_SEND_FAILED_EVENT 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 setup failed. If non-blocking mode is set and data was sent (on a one-to-many style socket), an SCTP_CANT_STR_ASSOC is accompanied by a series of SCTP_SEND_FAILED_EVENT events, one for each outstanding message.

sac_error: If the state was reached due to an 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 [RFC4960].

sac_outbound_streams and sac_inbound_streams: The maximum number of streams allowed in each direction is available in sac_outbound_streams and sac_inbound streams.

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

sac_info: If 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 Section 3.3.7 of the SCTP specification [RFC4960]. If sac_state is SCTP_COMM_UP or SCTP_RESTART, sac_info may contain an array of uint8_t describing the features that the current association supports. Features may include

SCTP_ASSOC_SUPPORTS_PR: Both endpoints support the protocol extension described in [RFC3758].

SCTP_ASSOC_SUPPORTS_AUTH: Both endpoints support the protocol extension described in [RFC4895].

SCTP_ASSOC_SUPPORTS_ASCONF: Both endpoints support the protocol extension described in [RFC5061].

SCTP_ASSOC_SUPPORTS_MULTIBUF: For a one-to-many style socket, the local endpoints use separate send and/or receive buffers for each SCTP association.

6.1.2. SCTP_PEER_ADDR_CHANGE

When a destination address of a multi-homed peer encounters a state change, a peer address change event is sent. The notification has the following format:

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

spc_type: This field should be set to SCTP_PEER_ADDR_CHANGE.

spc_flags: This field is currently unused.

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

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

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

SCTP_ADDR_AVAILABLE: This address is now reachable. This notification is provided whenever an address becomes 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. This notification is provided whenever an address becomes unreachable.

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 the primary destination address. This notification is provided whenever an address is made primary.

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

6.1.3. SCTP_REMOTE_ERROR

A remote peer may send an Operation 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 an SCTP_REMOTE_ERROR event. Please refer to the SCTP specification [RFC4960] and any extensions for a list of possible error formats. An SCTP error notification has the following 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[];
};
```

sre_type: This field should be set to SCTP_REMOTE_ERROR.

sre_flags: This field is currently unused.

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

sre_error: This value represents one of the Operation Error causes defined in the SCTP specification [RFC4960], in network byte order.

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

sre_data: This contains the ERROR chunk as defined in Section 3.3.10 of the SCTP specification [RFC4960].

6.1.4. SCTP_SEND_FAILED - DEPRECATED

Please note that this notification is deprecated. Use SCTP_SEND_FAILED_EVENT instead.

If SCTP cannot deliver a message, it can return back the message as a notification if the SCTP_SEND_FAILED event is enabled. The notification has the following format:

```
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[];
};
```

ssf_type: This field should be set to SCTP_SEND_FAILED.

ssf_flags: The flag value will take one of the following values:

SCTP_DATA_UNSENT: This value indicates that the data was never put on the wire.

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

ssf_error: This value represents the reason why the send failed, and if set, will be an SCTP protocol error code as defined in Section 3.3.10 of [RFC4960].

ssf_info: This field includes the ancillary data (struct sctp_sndrcvinfo) used to send the undelivered message. Regardless of whether ancillary data is used or not, the ssf_info.sinfo_flags field indicates whether the complete message or only part of the message is returned in ssf_data. If only part of the message is returned, it means that the part that is not present has been sent successfully to the peer.

If the complete message cannot be sent, the SCTP_DATA_NOT_FRAG flag is set in ssf_info.sinfo_flags. If the first part of the message is sent successfully, SCTP_DATA_LAST_FRAG is set. This means that the tail end of the message is returned in ssf_data.

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

ssf_data: The undelivered message or part of the undelivered message will be present in the ssf_data field. Note that the ssf_info.sinfo_flags field as noted above should be used to determine whether a complete message or just a piece of the message is present. Note that only user data is present in this field; any chunk headers or SCTP common headers must be removed by the SCTP stack.

6.1.5. 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: This field should be set to SCTP_SHUTDOWN_EVENT.

sse_flags: This field is currently unused.

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

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

6.1.6. SCTP_ADAPTATION_INDICATION

When a peer sends an Adaptation Layer Indication parameter as described in [RFC5061], SCTP delivers this notification to inform the application about the peer's adaptation layer indication.

```
struct sctp_adaptation_event {
  uint16_t sai_type;
  uint16_t sai_flags;
  uint32_t sai_length;
  uint32_t sai_adaptation_ind;
  sctp_assoc_t sai_assoc_id;
};
```

sai_type: This field should be set to SCTP_ADAPTATION_INDICATION.

sai_flags: This field is currently unused.

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

sai_adaptation_ind: This field holds the bit array sent by the peer in the Adaptation Layer Indication parameter.

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

6.1.7. 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;
  uint32_t pdapi_stream;
  uint32_t pdapi_seq;
  sctp_assoc_t pdapi_assoc_id;
};
```

pdapi_type: This field should be set to SCTP_PARTIAL_DELIVERY_EVENT.

pdapi_flags: This field is currently unused.

pdapi_length: 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: This field holds the indication being sent to the application. Currently, there is only one defined value:

SCTP_PARTIAL_DELIVERY_ABORTED: This indicates that the partial delivery of a user message has been aborted. This happens, for example, if an association is aborted while a partial delivery is going on or the user message gets abandoned using PR-SCTP while the partial delivery of this message is going on.

pdapi_stream: This field holds the stream on which the partial delivery event happened.

pdapi_seq: This field holds the stream sequence number that was being partially delivered.

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

6.1.8. SCTP_AUTHENTICATION_EVENT

[RFC4895] defines an extension to authenticate SCTP messages. The following notification is used to report different events relating to the use of this extension.

```
struct sctp_authkey_event {
  uint16_t auth_type;
  uint16_t auth_flags;
  uint32_t auth_length;
  uint16_t auth_keynumber;
  uint32_t auth_indication;
  sctp_assoc_t auth_assoc_id;
};
```

auth_type: This field should be set to SCTP_AUTHENTICATION_EVENT.

auth_flags: This field is currently unused.

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

auth_keynumber: This field holds the key number for the affected key indicated in the event (depends on auth_indication).

auth_indication: This field holds the error or indication being reported. The following values are currently defined:

SCTP_AUTH_NEW_KEY: This report indicates that a new key has been made active (used for the first time by the peer) and is now the active key. The auth_keynumber field holds the user-specified key number.

SCTP_AUTH_NO_AUTH: This report indicates that the peer does not support SCTP authentication as defined in [RFC4895].

SCTP_AUTH_FREE_KEY: This report indicates that the SCTP implementation will no longer use the key identifier specified in auth_keynumber.

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

6.1.9. SCTP_SENDER_DRY_EVENT

When the SCTP stack has no more user data to send or retransmit, this notification is given to the user. Also, at the time when a user app subscribes to this event, if there is no data to be sent or retransmit, the stack will immediately send up this notification.

```
struct sctp_sender_dry_event {
  uint16_t sender_dry_type;
  uint16_t sender_dry_flags;
  uint32_t sender_dry_length;
  sctp_assoc_t sender_dry_assoc_id;
};
```

sender_dry_type: This field should be set to SCTP_SENDER_DRY_EVENT.

sender_dry_flags: This field is currently unused.

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

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

6.1.10. SCTP_NOTIFICATIONS_STOPPED_EVENT

SCTP notifications, when subscribed to, are reliable. They are always delivered as long as there is space in the socket receive buffer. However, if an implementation experiences a notification storm, it may run out of socket buffer space. When this occurs, it may wish to disable notifications. If the implementation chooses to do this, it will append a final notification SCTP_NOTIFICATIONS_STOPPED_EVENT. This notification is a union sctp_notification, where only the sctp_tlv structure (see the union above) is used. It only contains this type in the sn_type field, the sn_length field set to the size of an sctp_tlv structure, and the sn_flags set to 0. If an application receives this notification, it will need to re-subscribe to any notifications of interest to it, except for the sctp_data_io_event (note that SCTP_EVENTS is deprecated).

An endpoint is automatically subscribed to this event as soon as it is subscribed to any event other than data io events.

6.1.11. SCTP_SEND_FAILED_EVENT

If SCTP cannot deliver a message, it can return back the message as a notification if the SCTP_SEND_FAILED_EVENT event is enabled. The notification has the following format:

```
struct sctp_send_failed_event {
  uint16_t ssfe_type;
  uint16_t ssfe_flags;
  uint32_t ssfe_length;
  uint32_t ssfe_error;
  struct sctp_sndinfo ssfe_info;
  sctp_assoc_t ssfe_assoc_id;
  uint8_t  ssfe_data[];
};
```

ssfe_type: This field should be set to SCTP_SEND_FAILED_EVENT.

ssfe_flags: The flag value will take one of the following values:

SCTP_DATA_UNSENT: This value indicates that the data was never put on the wire.

SCTP_DATA_SENT: This value 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.

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

ssfe_error: This value represents the reason why the send failed, and if set, will be an SCTP protocol error code as defined in Section 3.3.10 of [RFC4960].

ssfe_info: This field includes the ancillary data (struct sctp_sndinfo) used to send the undelivered message. Regardless of whether ancillary data is used or not, the ssfe_info.sinfo_flags field indicates whether the complete message or only part of the message is returned in ssf_data. If only part of the message is returned, it means that the part that is not present has been sent successfully to the peer.

If the complete message cannot be sent, the SCTP_DATA_NOT_FRAG flag is set in ssfe_info.sinfo_flags. If the first part of the message is sent successfully, SCTP_DATA_LAST_FRAG is set. This means that the tail end of the message is returned in ssf_data.

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

ssfe_data: The undelivered message or part of the undelivered message will be present in the ssf_data field. Note that the ssf_info.sinfo_flags field as noted above should be used to determine whether a complete message or just a piece of the message is present. Note that only user data is present in this field; any chunk headers or SCTP common headers must be removed by the SCTP stack.

6.2. Notification Interest Options

6.2.1. SCTP_EVENTS Option - DEPRECATED

Please note that this option is deprecated. Use the SCTP_EVENT option described in Section 6.2.2 instead.

To receive SCTP event notifications, an application registers its interest by setting the SCTP_EVENTS socket option. The application then uses recvmsg() to retrieve notifications. A notification is stored in the data part (msg_iov) of the msghdr structure. The socket option uses 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_adaptation_layer_event;
  uint8_t sctp_authentication_event;
  uint8_t sctp_sender_dry_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. Setting the flag to 0 will disable the reception of the message control information. Note that this flag is not really a notification and is stored in the ancillary data (msg_control), not in the data part (msg_iov).

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.

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.

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.

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.

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.

sctp_partial_delivery_event: Setting this flag to 1 will enable the reception of partial delivery event notifications. Setting the flag to 0 will disable partial delivery event notifications.

sctp_adaptation_layer_event: Setting this flag to 1 will enable the reception of adaptation layer event notifications. Setting the flag to 0 will disable adaptation layer event notifications.

sctp_authentication_event: Setting this flag to 1 will enable the reception of authentication layer event notifications. Setting the flag to 0 will disable authentication layer event notifications.

sctp_sender_dry_event: Setting this flag to 1 will enable the reception of sender dry event notifications. Setting the flag to 0 will disable sender dry event notifications.

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 events;

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

  events.sctp_data_io_event = 1;
  events.sctp_association_event = 1;

  setsockopt(sd, IPPROTO_SCTP, SCTP_EVENTS, &events, sizeof(events));
}
```

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 only for the single association bound to the file descriptor.

By default, both the one-to-one style and the one-to-many style socket do not subscribe to any notification.

6.2.2. SCTP_EVENT Option

The SCTP_EVENTS socket option has one issue for future compatibility. As new features are added, the structure (sctp_event_subscribe) must be expanded. This can cause an application binary interface (ABI) issue unless an implementation has added padding at the end of the structure. To avoid this problem, SCTP_EVENTS has been deprecated and a new socket option SCTP_EVENT has taken its place. The option is used with the following structure:

```
struct sctp_event {
        sctp_assoc_t se_assoc_id;
        uint16_t     se_type;
        uint8_t      se_on;
};
```

se_assoc_id: The se_assoc_id field is ignored for one-to-one style sockets. For one-to-many style sockets, this field can be a particular association identifier or SCTP_{FUTURE|CURRENT|ALL}_ASSOC.

se_type: The se_type field can be filled with any value that would show up in the respective sn_type field (in the sctp_tlv structure of the notification).

se_on: The se_on field is set to 1 to turn on an event and set to 0 to turn off an event.

To use this option, the user fills in this structure and then calls setsockopt() to turn on or off an individual event. The following is an example use of this option:

```
{
  struct sctp_event event;

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

  event.se_assoc_id = SCTP_FUTURE_ASSOC;
  event.se_type = SCTP_SENDER_DRY_EVENT;
  event.se_on = 1;
  setsockopt(sd, IPPROTO_SCTP, SCTP_EVENT, &event, sizeof(event));
}
```

By default, both the one-to-one style and the one-to-many style socket do not subscribe to any notification.

7. Common Operations for Both Styles

7.1. send(), recv(), sendto(), and 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 function prototypes are

```
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);
```

and the arguments are

sd: The socket descriptor of an SCTP endpoint.

msg: The message to be sent.

len: The size of the message or the size of the 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 cannot 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 that 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. Sending a message using send() or sendto() is atomic unless explicit EOR marking is enabled on the socket specified by sd. Using sendto() on a non-connected one-to-one style socket for implicit connection setup may or may not work, depending on the SCTP implementation. recv() and recvfrom() cannot distinguish message boundaries (i.e., there is no way to observe the MSG_EOR flag to detect partial delivery).

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

Note that if an application calls a send() or sendto() function with no user 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 [RFC4960].

7.2. setsockopt() and getsockopt()

Applications use setsockopt() and getsockopt() to set or retrieve socket options. Socket options are used to change the default behavior of socket calls. They are described in Section 8.

The function prototypes are

```
int getsockopt(int sd,
               int level,
               int optname,
               void *optval,
               socklen_t *optlen);
```

and

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

and the arguments are

sd: The socket descriptor.

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).

These functions return 0 on success and -1 in case of an error.

All socket options set on a one-to-one style listening socket also apply to all future accepted sockets. For one-to-many style sockets, often a socket option will pass a structure that includes an assoc_id field. This field can be filled with the association identifier of a particular association and unless otherwise specified can be filled with one of the following constants:

SCTP_FUTURE_ASSOC: Specifies that only future associations created after this socket option will be affected by this call.

SCTP_CURRENT_ASSOC: Specifies that only currently existing associations will be affected by this call, and future associations will still receive the previous default value.

SCTP_ALL_ASSOC: Specifies that all current and future associations will be affected by this call.

7.3. read() and write()

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

7.4. getsockname()

Applications use getsockname() to retrieve the locally bound socket address of the specified socket. This is especially useful if the caller let SCTP choose a local port. This call is for single-homed endpoints. It does not work well with multi-homed endpoints. See Section 9.5 for a multi-homed version of the call.

The function prototype is

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

and the arguments are

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 the address here. On return, this is set to the length of the returned address.

getsockname() returns 0 on success and -1 in case of an error.

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.

7.5. Implicit Association Setup

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 no association exists between the sender and the intended receiver (identified by the address passed either in the msg_name field of the msghdr structure in the sendmsg() call or the dest_addr field in the sendto() call), the SCTP stack will automatically set up an association to the intended receiver.

Upon successful association setup, an 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 3.1.4).

Note that if the SCTP stack at the sender side supports bundling, the first user message may be bundled with the COOKIE ECHO message [RFC4960].

When the SCTP stack sets up a new association implicitly, the SCTP_INIT type ancillary data may also be passed along (see Section 5.3.1 for details of the data structures) to change some parameters used in setting up a 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() calls.

8. Socket Options

The following subsection 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 to which peer address the option should be applied.

For the one-to-many style sockets, an sctp_assoc_t (association identifier) parameter is used to identify 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 9.2), 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, the options will be applied to all associations (similar to using SCTP_ALL_ASSOC as the association identifier) belonging to the socket. For the one-to-one style, these options will be applied to all peer addresses of the association controlled by the socket. Applications should be 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 into 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() call must be used to set any writable option.

For one-to-many style sockets, id specifies the association to query. For one-to-one style sockets, id is ignored. For one-to-many style sockets, any association identifier in the structure provided as arg is ignored, and id takes precedence.

Note that SCTP_CURRENT_ASSOC and SCTP_ALL_ASSOC cannot be used with sctp_opt_info() or in getsockopt() calls. Using them will result in an error (returning -1 and errno set to EINVAL). SCTP_FUTURE_ASSOC can be used to query information for future associations.

The field 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_PRIMARY_ADDR

SCTP_PEER_ADDR_PARAMS

SCTP_DEFAULT_SEND_PARAM

SCTP_MAX_SEG

SCTP_AUTH_ACTIVE_KEY

SCTP_DELAYED_SACK

SCTP_MAX_BURST

SCTP_CONTEXT

SCTP_EVENT

SCTP_DEFAULT_SNDINFO

SCTP_DEFAULT_PRINFO

SCTP_STATUS

SCTP_GET_PEER_ADDR_INFO

SCTP_PEER_AUTH_CHUNKS

SCTP_LOCAL_AUTH_CHUNKS

The arg field is an option-specific structure buffer provided by the caller. See the rest of this section 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.

8.1. Read/Write Options

8.1.1. Retransmission Timeout Parameters (SCTP_RTOINFO)

The protocol parameters used to initialize and limit the retransmission timeout (RTO) are tunable. See [RFC4960] 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_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, the application may fill in an association identifier or SCTP_FUTURE_ASSOC. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in srto_assoc_id.

srto_initial: This parameter contains the initial RTO value.

srto_max and srto_min: These parameters contain the maximum and minimum bounds for all RTOs.

All times are given 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.

8.1.2. Association Parameters (SCTP_ASSOCINFO)

This option is used to both examine and set various association and endpoint parameters. See [RFC4960] for more information on how these parameters are used.

The following structure is used to access and modify these 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_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, the application may fill in an association identifier or SCTP_FUTURE_ASSOC. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in sasoc_assoc_id.

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

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

sasoc_peer_rwnd: This parameter holds the current value of the peer's rwnd (reported in the last selective acknowledgment (SACK)) minus any outstanding data (i.e., data in flight).

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

sasoc_cookie_life: This parameter is the association's cookie life value used when issuing cookies.

The value of sasoc_peer_rwnd is meaningless when examining endpoint information (i.e., it is only valid when examining information on a specific association).

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

The values of 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's maximum retransmission parameter to more than the sum of all maximum retransmission parameters, setsockopt() may return an error. The reason for this, from Section 8.2 of [RFC4960], is as follows:

> Note: When configuring the SCTP endpoint, the user should avoid having the value of 'Association.Max.Retrans' (sasoc_maxrxt in this option) larger than the summation of the 'Path.Max.Retrans' (see spp_pathmaxrxt in Section 8.1.12) of all of the destination addresses for the remote endpoint. Otherwise, all of the destination addresses may become inactive while the endpoint still considers the peer endpoint reachable.

8.1.3. 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 5.3.1. 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 affected by the change).

8.1.4. SO_LINGER

An application can use this option to perform the SCTP ABORT primitive. This option affects all associations related to the socket.

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. Please note that the time unit is in seconds, according to POSIX, but might be different on specific platforms. If the graceful shutdown phase does not finish during this period, close() will return, but the graceful shutdown phase will continue in the system.

Note that this is a socket-level option, not an SCTP-level option. When using this option, an application must specify a level of SOL_SOCKET in the call.

8.1.5. SCTP_NODELAY

This option turns 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. In particular, not using any Nagle-like algorithm might reduce the bundling of small user messages in cases where this would require an additional delay.

Turning this option on disables any Nagle-like algorithm.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.6. SO_RCVBUF

This option sets the receive buffer size in octets. For SCTP one-to-one style sockets, this option controls the receiver window size. For one-to-many style sockets, the meaning is implementation dependent. It might control the receive buffer for each association bound to the socket descriptor, or it might control the receive buffer for the whole socket. This option expects an integer.

Note that this is a socket-level option, not an SCTP-level option. When using this option, an application must specify a level of SOL_SOCKET in the call.

8.1.7. SO_SNDBUF

This option sets the send buffer size. For SCTP one-to-one style sockets, this option 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 3.3) bound to the socket descriptor used in the setsockopt() or getsockopt() call. The option applies to each association's window size separately. This option expects an integer.

Note that this is a socket-level option, not an SCTP-level option. When using this option, an application must specify a level of SOL_SOCKET in the call.

8.1.8. 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 using the graceful shutdown procedure. An idle association 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 association should be performed; this is the default value. This option expects an integer defining the number of seconds of idle time before an association is closed.

An application using this option should enable the ability to receive the association change notification. This is the only mechanism by which an application is informed about the closing of an association. After an association is closed, the association identifier 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 endpoint.

8.1.9. Set Primary Address (SCTP_PRIMARY_ADDR)

This option requests that the local SCTP stack uses the enclosed peer address as the association's 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_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, it identifies the association for this request. Note that the special sctp_assoc_t SCTP_{FUTURE|ALL|CURRENT}_ASSOC are not allowed.

ssp_addr: This parameter is the address to set as primary. No wildcard address is allowed.

8.1.10. Set Adaptation Layer Indicator (SCTP_ADAPTATION_LAYER)

This option requests that the local endpoint set the specified Adaptation Layer Indication parameter for all future INIT and INIT-ACK exchanges.

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

```
struct sctp_setadaptation {
  uint32_t   ssb_adaptation_ind;
};
```

ssb_adaptation_ind: The adaptation layer indicator that will be included in any outgoing Adaptation Layer Indication parameter.

8.1.11. Enable/Disable Message Fragmentation (SCTP_DISABLE_FRAGMENTS)

This option is an on/off flag and is passed as an integer, where a non-zero is on and a zero is off. If enabled, no SCTP message fragmentation will be performed. The effect of enabling this option

is that if a message being sent exceeds the current Path MTU (PMTU) size, the message will not be sent and instead an error will be indicated to the user. If this option is disabled (the default), then a message exceeding the size of the PMTU will be fragmented and reassembled by the peer.

8.1.12. 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_flags;
  uint32_t spp_ipv6_flowlabel;
  uint8_t  spp_dscp;
};
```

spp_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, the application may fill in an association identifier or SCTP_FUTURE_ASSOC for this query. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in spp_assoc_id.

spp_address: This specifies which address is of interest. If a wildcard address is provided, it applies to all current and future paths.

spp_hbinterval: This contains the value of the heartbeat interval, in milliseconds (HB.Interval in [RFC4960]). Note that unless the spp_flags field is set to SPP_HB_ENABLE, the value of this field is ignored. Note also that a value of zero indicates that the current setting should be left unchanged. To set an actual value of zero, the SPP_HB_TIME_IS_ZERO flag should be used. Even when it is set to 0, it does not mean that SCTP will continuously send out heartbeats, since the actual interval also includes the current RTO and jitter (see Section 8.3 of [RFC4960]).

spp_pathmaxrxt: This contains the maximum number of retransmissions before this address shall be considered unreachable. Note that a value of zero indicates that the current setting should be left unchanged.

spp_pathmtu: This field contains the current Path MTU of the peer address. It is the number of bytes available in an SCTP packet for chunks. Providing a value of 0 does not change the current setting. If a positive value is provided and SPP_PMTUD_DISABLE is set in the spp_flags field, the given value is used as the Path MTU. If SPP_PMTUD_ENABLE is set in the spp_flags field, the spp_pathmtu field is ignored.

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

SPP_HB_ENABLE: This field enables heartbeats on the specified address.

SPP_HB_DISABLE: This field disables heartbeats on the specified address. Note that SPP_HB_ENABLE and SPP_HB_DISABLE are mutually exclusive; only one of these two should be specified. Enabling both fields will yield undetermined results.

SPP_HB_DEMAND: This field requests that a user-initiated heartbeat be made immediately. This must not be used in conjunction with a wildcard address.

SPP_HB_TIME_IS_ZERO: This field specifies that the time for heartbeat delay is to be set to 0 milliseconds.

SPP_PMTUD_ENABLE: This field will enable PMTU discovery on the specified address.

SPP_PMTUD_DISABLE: This field will disable PMTU discovery on the specified address. Note that if the address field is empty, then all addresses on the association are affected. Note also that SPP_PMTUD_ENABLE and SPP_PMTUD_DISABLE are mutually exclusive. Enabling both fields will yield undetermined results.

SPP_IPV6_FLOWLABEL: Setting this flag enables the setting of the IPV6 flow label value. The value is contained in the spp_ipv6_flowlabel field.

Upon retrieval, this flag will be set to indicate that the spp_ipv6_flowlabel field has a valid value returned. If a specific destination address is set (in the spp_address field), then the value returned is that of the address. If just an association is specified (and no address), then the association's default flow label is returned. If neither an association nor a destination is specified, then the socket's default flow label is returned. For non-IPv6 sockets, this flag will be left cleared.

SPP_DSCP: Setting this flag enables the setting of the Differentiated Services Code Point (DSCP) value associated with either the association or a specific address. The value is obtained in the spp_dscp field.

Upon retrieval, this flag will be set to indicate that the spp_dscp field has a valid value returned. If a specific destination address is set when called (in the spp_address field), then that specific destination address's DSCP value is returned. If just an association is specified, then the association's default DSCP is returned. If neither an association nor a destination is specified, then the socket's default DSCP is returned.

spp_ipv6_flowlabel: This field is used in conjunction with the SPP_IPV6_FLOWLABEL flag and contains the IPv6 flow label. The 20 least significant bits are used for the flow label. This setting has precedence over any IPv6-layer setting.

spp_dscp: This field is used in conjunction with the SPP_DSCP flag and contains the DSCP. The 6 most significant bits are used for the DSCP. This setting has precedence over any IPv4- or IPv6-layer setting.

Please note that changing the flow label or DSCP value will affect all packets sent by the SCTP stack after setting these parameters. The flow label might also be set via the sin6_flowinfo field of the sockaddr_in6 structure.

8.1.13. Set Default Send Parameters (SCTP_DEFAULT_SEND_PARAM) - DEPRECATED

Please note that this option is deprecated. SCTP_DEFAULT_SNDINFO (Section 8.1.31) should be used instead.

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 the sctp_sndrcvinfo structure (defined in Section 5.3.2) to this call. The input parameters accepted by this call include sinfo_stream, sinfo_flags, sinfo_ppid, sinfo_context, and sinfo_timetolive. The sinfo_flags field is composed of a bitwise OR of SCTP_UNORDERED, SCTP_EOF, and SCTP_SENDALL. The sinfo_assoc_id field specifies the association to which to apply the parameters. For a one-to-many style socket, any of the predefined constants are also allowed in this field. The field is ignored for one-to-one style sockets.

8.1.14. Set Notification and Ancillary Events (SCTP_EVENTS) - DEPRECATED

This socket option is used to specify various notifications and ancillary data the user wishes to receive. Please see Section 6.2.1 for a full description of this option and its usage. Note that this option is considered deprecated and is present for backward compatibility. New applications should use the SCTP_EVENT option. See Section 6.2.2 for a full description of that option as well.

8.1.15. Set/Clear IPv4 Mapped Addresses (SCTP_I_WANT_MAPPED_V4_ADDR)

This socket option is a boolean flag that turns on or off the mapping of IPv4 addresses. If this option is turned on, then IPv4 addresses will be mapped to IPv6 representation. If this option is turned off, then no mapping will be done of IPv4 addresses, and a user will receive both PF_INET6 and PF_INET type addresses on the socket. See [RFC3542] for more details on mapped IPv6 addresses.

If this socket option is used on a socket of type PF_INET, an error is returned.

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

8.1.16. Get or Set the Maximum Fragmentation Size (SCTP_MAXSEG)

This option will get or set the maximum size to put in any outgoing SCTP DATA chunk. If a message is larger than this maximum 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 default value for this option is '0', which indicates that the user is not limiting fragmentation and only the PMTU will affect SCTP's choice of DATA chunk size. Note also that

values set larger than the maximum size of an IP datagram will effectively let SCTP control fragmentation (i.e., the same as setting this option to 0).

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

```
struct sctp_assoc_value {
  sctp_assoc_t assoc_id;
  uint32_t assoc_value;
};
```

assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, this parameter indicates upon which association the user is performing an action. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in assoc_id.

assoc_value: This parameter specifies the maximum size in bytes.

8.1.17. Get or Set the List of Supported HMAC Identifiers (SCTP_HMAC_IDENT)

This option gets or sets the list of Hashed Message Authentication Code (HMAC) algorithms that the local endpoint requires the peer to use.

The following structure is used to get or set these identifiers:

```
struct sctp_hmacalgo {
  uint32_t shmac_number_of_idents;
  uint16_t shmac_idents[];
};
```

shmac_number_of_idents: This field gives the number of elements present in the array shmac_idents.

shmac_idents: This parameter contains an array of HMAC identifiers that the local endpoint is requesting the peer to use, in priority order. The following identifiers are valid:

* SCTP_AUTH_HMAC_ID_SHA1

* SCTP_AUTH_HMAC_ID_SHA256

Note that the list supplied must include SCTP_AUTH_HMAC_ID_SHA1 and may include any of the other values in its preferred order (lowest list position has the highest preference in algorithm selection).

Note also that the lack of SCTP_AUTH_HMAC_ID_SHA1, or the inclusion of an unknown HMAC identifier (including optional identifiers unknown to the implementation), will cause the set option to fail and return an error.

8.1.18. Get or Set the Active Shared Key (SCTP_AUTH_ACTIVE_KEY)

This option will get or set the active shared key to be used to build the association shared key.

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

```
struct sctp_authkeyid {
  sctp_assoc_t scact_assoc_id;
  uint16_t scact_keynumber;
};
```

scact_assoc_id: This parameter sets the active key of the specified association. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can be used. For one-to-one style sockets, this parameter is ignored. Note, however, that this option will set the active key on the association if the socket is connected; otherwise, this option will set the default active key for the endpoint.

scact_keynumber: This parameter is the shared key identifier that the application is requesting to become the active shared key to be used for sending authenticated chunks. The key identifier must correspond to an existing shared key. Note that shared key identifier '0' defaults to a null key.

When used with setsockopt(), the SCTP implementation must use the indicated shared key identifier for all messages being given to an SCTP implementation via a send call after the setsockopt() call, until changed again. Therefore, the SCTP implementation must not bundle user messages that should be authenticated using different shared key identifiers.

Initially, the key with key identifier 0 is the active key.

8.1.19. Get or Set Delayed SACK Timer (SCTP_DELAYED_SACK)

This option will affect the way delayed SACKs are performed. This option allows the application to get or set the delayed SACK time, in milliseconds. It also allows changing the delayed SACK frequency. Changing the frequency to 1 disables the delayed SACK algorithm. Note that if sack_delay or sack_freq is 0 when setting this option, the current values will remain unchanged.

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

```
struct sctp_sack_info {
  sctp_assoc_t sack_assoc_id;
  uint32_t sack_delay;
  uint32_t sack_freq;
};
```

sack_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, this parameter indicates upon which association the user is performing an action. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can also be used.

sack_delay: This parameter contains the number of milliseconds the user is requesting that the delayed SACK timer be set to. Note that this value is defined in [RFC4960] to be between 200 and 500 milliseconds.

sack_freq: This parameter contains the number of packets that must be received before a SACK is sent without waiting for the delay timer to expire. The default value is 2; setting this value to 1 will disable the delayed SACK algorithm.

8.1.20. Get or Set Fragmented Interleave (SCTP_FRAGMENT_INTERLEAVE)

Fragmented interleave controls how the presentation of messages occurs for the message receiver. There are three levels of fragment interleave defined. Two of the levels affect one-to-one style sockets, while one-to-many style sockets are affected by all three levels.

This option takes an integer value. It can be set to a value of 0, 1, or 2. Attempting to set this level to other values will return an error.

Setting the three levels provides the following receiver interactions:

level 0: Prevents the interleaving of any messages. This means that when a partial delivery begins, no other messages will be received except the message being partially delivered. If another message arrives on a different stream (or association) that could be delivered, it will be blocked waiting for the user to read all of the partially delivered message.

level 1: Allows interleaving of messages that are from different associations. For one-to-one style sockets, level 0 and level 1 thus have the same meaning, since a one-to-one style socket always receives messages from the same association. Note that setting a one-to-many style socket to this level may cause multiple partial deliveries from different associations, but for any given association, only one message will be delivered until all parts of a message have been delivered. This means that one large message, being read with an association identifier of "X", will block other messages from association "X" from being delivered.

level 2: Allows complete interleaving of messages. This level requires that the sender not only carefully observe the peer association identifier (or address) but also pay careful attention to the stream number. With this option enabled, a partially delivered message may begin being delivered for association "X" stream "Y", and the next subsequent receive may return a message from association "X" stream "Z". Note that no other messages would be delivered for association "X" stream "Y" until all of stream "Y"'s partially delivered message was read. Note that this option also affects one-to-one style sockets. Also note that for one-to-many style sockets, not only another stream's message from the same association may be delivered upon the next receive, but some other association's message may also be delivered upon the next receive.

An implementation should default one-to-many style sockets to level 1, because otherwise, it is possible that a peer could begin sending a partial message and thus block all other peers from sending data. However, a setting of level 2 requires that the application not only be aware of the association (via the association identifier or peer's address) but also the stream number. The stream number is not present unless the user has subscribed to the sctp_data_io_event (see Section 6.2), which is deprecated, or has enabled the SCTP_RECVRCVINFO socket option (see Section 8.1.29). This is also why we recommend that one-to-one style sockets be defaulted to level 0 (level 1 for one-to-one style sockets has no effect). Note that an implementation should return an error if an application attempts to set the level to 2 and has not subscribed to the sctp_data_io_event event, which is deprecated, or has enabled the SCTP_RECVRCVINFO socket option.

For applications that have subscribed to events, those events appear in the normal socket buffer data stream. This means that unless the user has set the fragmentation interleave level to 0, notifications may also be interleaved with partially delivered messages.

8.1.21. Set or Get the SCTP Partial Delivery Point (SCTP_PARTIAL_DELIVERY_POINT)

This option will set or get the SCTP partial delivery point. This point is the size of a message where the partial delivery API will be invoked to help free up rwnd space for the peer. Setting this to a lower value will cause partial deliveries to happen more often. This option expects an integer that sets or gets the partial delivery point in bytes. Note also that the call will fail if the user attempts to set this value larger than the socket receive buffer size.

Note that any single message having a length smaller than or equal to the SCTP partial delivery point will be delivered in a single read call as long as the user-provided buffer is large enough to hold the message.

8.1.22. Set or Get the Use of Extended Receive Info (SCTP_USE_EXT_RCVINFO) - DEPRECATED

This option will enable or disable the use of the extended version of the sctp_sndrcvinfo structure. If this option is disabled, then the normal sctp_sndrcvinfo structure is returned in all receive message calls. If this option is enabled, then the sctp_extrcvinfo structure is returned in all receive message calls. The default is off.

Note that the sctp_extrcvinfo structure is never used in any send call.

This option is present for compatibility with older applications and is deprecated. Future applications should use SCTP_NXTINFO to retrieve this same information via ancillary data.

8.1.23. Set or Get the Auto ASCONF Flag (SCTP_AUTO_ASCONF)

This option will enable or disable the use of the automatic generation of ASCONF chunks to add and delete addresses to an existing association. Note that this option has two caveats, namely a) it only affects sockets that are bound to all addresses available to the SCTP stack, and b) the system administrator may have an overriding control that turns the ASCONF feature off no matter what setting the socket option may have.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.24. Set or Get the Maximum Burst (SCTP_MAX_BURST)

This option will allow a user to change the maximum burst of packets that can be emitted by this association. Note that the default value is 4, and some implementations may restrict this setting so that it can only be lowered to positive values.

To set or get this option, the user fills in the following structure:

```
struct sctp_assoc_value {
  sctp_assoc_t assoc_id;
  uint32_t assoc_value;
};
```

assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, this parameter indicates upon which association the user is performing an action. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can also be used.

assoc_value: This parameter contains the maximum burst. Setting the value to 0 disables burst mitigation.

8.1.25. Set or Get the Default Context (SCTP_CONTEXT)

The context field in the sctp_sndrcvinfo structure is normally only used when a failed message is retrieved holding the value that was sent down on the actual send call. This option allows the setting, on an association basis, of a default context that will be received on reading messages from the peer. This is especially helpful for an application when using one-to-many style sockets to keep some reference to an internal state machine that is processing messages on the association. Note that the setting of this value only affects received messages from the peer and does not affect the value that is saved with outbound messages.

To set or get this option, the user fills in the following structure:

```
struct sctp_assoc_value {
  sctp_assoc_t assoc_id;
  uint32_t assoc_value;
};
```

assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, this parameter indicates upon which association the user is performing an action. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can also be used.

assoc_value: This parameter contains the context.

8.1.26. Enable or Disable Explicit EOR Marking (SCTP_EXPLICIT_EOR)

This boolean flag is used to enable or disable explicit end of record (EOR) marking. When this option is enabled, a user may make multiple send system calls to send a record and must indicate that they are finished sending a particular record by including the SCTP_EOR flag. If this boolean flag is disabled, then each individual send system call is considered to have an SCTP_EOR indicator set on it implicitly without the user having to explicitly add this flag. The default is off.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.27. Enable SCTP Port Reusage (SCTP_REUSE_PORT)

This option only supports one-to-one style SCTP sockets. If used on a one-to-many style SCTP socket, an error is indicated.

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

This socket option must not be used after calling bind() or sctp_bindx() for a one-to-one style SCTP socket. If using bind() or sctp_bindx() on a socket with the SCTP_REUSE_PORT option, all other SCTP sockets bound to the same port must have set the SCTP_REUSE_PORT option. Calling bind() or sctp_bindx() for a socket without having set the SCTP_REUSE_PORT option will fail if there are other sockets bound to the same port. At most one socket being bound to the same port may be listening.

It should be noted that the behavior of the socket-level socket option to reuse ports and/or addresses for SCTP sockets is unspecified.

8.1.28. Set Notification Event (SCTP_EVENT)

This socket option is used to set a specific notification option. Please see Section 6.2.2 for a full description of this option and its usage.

8.1.29. Enable or Disable the Delivery of SCTP_RCVINFO as Ancillary Data (SCTP_RECVRCVINFO)

Setting this option specifies that SCTP_RCVINFO (defined in Section 5.3.5) is returned as ancillary data by recvmsg().

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.30. Enable or Disable the Delivery of SCTP_NXTINFO as Ancillary Data (SCTP_RECVNXTINFO)

Setting this option specifies that SCTP_NXTINFO (defined in Section 5.3.6) is returned as ancillary data by recvmsg().

This option expects an integer boolean flag, where a non-zero value turns on the option, and a zero value turns off the option.

8.1.31. Set Default Send Parameters (SCTP_DEFAULT_SNDINFO)

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_sndinfo structure. The application that wishes to use this socket option simply passes the sctp_sndinfo structure (defined in Section 5.3.4) to this call. The input parameters accepted by this call include snd_sid, snd_flags, snd_ppid, and snd_context. The snd_flags parameter is composed of a bitwise OR of SCTP_UNORDERED, SCTP_EOF, and SCTP_SENDALL. The snd_assoc_id field specifies the association to which to apply the parameters. For a one-to-many style socket, any of the predefined constants are also allowed in this field. The field is ignored for one-to-one style sockets.

8.1.32. Set Default PR-SCTP Parameters (SCTP_DEFAULT_PRINFO)

This option sets and gets the default parameters for PR-SCTP. They can be overwritten by specific information provided in send calls.

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

```
struct sctp_default_prinfo {
  uint16_t pr_policy;
  uint32_t pr_value;
  sctp_assoc_t pr_assoc_id;
};
```

pr_policy: This field is the same as that described in Section 5.3.7.

pr_value: This field is the same as that described in Section 5.3.7.

pr_assoc_id: This field is ignored for one-to-one style sockets. For one-to-many style sockets, pr_assoc_id can be a particular association identifier or SCTP_{FUTURE|CURRENT|ALL}_ASSOC.

8.2. Read-Only Options

The options defined in this subsection are read-only. Using this option in a setsockopt() call will result in an error indicating EOPNOTSUPP.

8.2.1. Association Status (SCTP_STATUS)

Applications can retrieve current status information about an association, including association state, peer receiver window size, number of unacknowledged 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_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, it holds the identifier for the association. All notifications for a given association have the same association identifier. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC cannot be used.

sstat_state: This contains the association's current state, i.e., 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 unacknowledged DATA chunks.

sstat_penddata: This is the number of DATA chunks pending receipt.

sstat_instrms: This is the number of streams that the peer will be using outbound.

sstat_outstrms: This is the number of outbound streams that the endpoint is allowed to use.

sstat_fragmentation_point: This is the size at which SCTP fragmentation will occur.

sstat_primary: This is information on the current primary peer address.

To access these status values, the application calls getsockopt() with the option name SCTP_STATUS.

8.2.2. 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_assoc_id: This parameter is ignored for one-to-one style sockets.

For one-to-many style sockets, this field may be filled by the application, and if so, this field will have priority in looking up the association instead of using the address specified in spinfo_address. Note that if the address does not belong to the association specified, then this call will fail. If the application does not fill in the spinfo_assoc_id, then the address will be used to look up the association, and on return, this field will have the valid association identifier. In other words, this call can be used to translate an address into an association identifier. Note that the predefined constants are not allowed for this option.

spinfo_address: This is filled by the application and contains the peer address of interest.

spinfo_state: This contains the peer address's state:

SCTP_UNCONFIRMED: This is the initial state of a peer address.

SCTP_ACTIVE: This state is entered the first time after path verification. It can also be entered if the state is SCTP_INACTIVE and the path supervision detects that the peer address is reachable again.

SCTP_INACTIVE: This state is entered whenever a path failure is detected.

spinfo_cwnd: This contains the peer address's current congestion window.

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

spinfo_rto: This contains the peer address's current retransmission timeout value in milliseconds.

spinfo_mtu: This is the current Path MTU of the peer address. It is the number of bytes available in an SCTP packet for chunks.

8.2.3. Get the List of Chunks the Peer Requires to Be Authenticated (SCTP_PEER_AUTH_CHUNKS)

This option gets a list of chunk types (see [RFC4960]) for a specified association that the peer requires to be received authenticated only.

The following structure is used to access these parameters:

```
struct sctp_authchunks {
  sctp_assoc_t gauth_assoc_id;
  uint32_t gauth_number_of_chunks
  uint8_t  gauth_chunks[];
};
```

gauth_assoc_id: This parameter indicates for which association the user is requesting the list of peer-authenticated chunks. For one-to-one style sockets, this parameter is ignored. Note that the predefined constants are not allowed with this option.

gauth_number_of_chunks: This parameter gives the number of elements in the array gauth_chunks.

gauth_chunks: This parameter contains an array of chunk types that the peer is requesting to be authenticated. If the passed-in buffer size is not large enough to hold the list of chunk types, ENOBUFS is returned.

8.2.4. Get the List of Chunks the Local Endpoint Requires to Be Authenticated (SCTP_LOCAL_AUTH_CHUNKS)

This option gets a list of chunk types (see [RFC4960]) for a specified association that the local endpoint requires to be received authenticated only.

The following structure is used to access these parameters:

```
struct sctp_authchunks {
  sctp_assoc_t gauth_assoc_id;
  uint32_t gauth_number_of_chunks;
  uint8_t  gauth_chunks[];
};
```

gauth_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, the application may fill in an association identifier or SCTP_FUTURE_ASSOC. It is an error to use SCTP_{CURRENT|ALL}_ASSOC in gauth_assoc_id.

gauth_number_of_chunks: This parameter gives the number of elements in the array gauth_chunks.

gauth_chunks: This parameter contains an array of chunk types that the local endpoint is requesting to be authenticated. If the passed-in buffer is not large enough to hold the list of chunk types, ENOBUFS is returned.

8.2.5. Get the Current Number of Associations (SCTP_GET_ASSOC_NUMBER)

This option gets the current number of associations that are attached to a one-to-many style socket. The option value is an uint32_t. Note that this number is only a snapshot. This means that the number of associations may have changed when the caller gets back the option result.

For a one-to-one style socket, this socket option results in an error.

8.2.6. Get the Current Identifiers of Associations (SCTP_GET_ASSOC_ID_LIST)

This option gets the current list of SCTP association identifiers of the SCTP associations handled by a one-to-many style socket.

The option value has the structure

```
struct sctp_assoc_ids {
  uint32_t gaids_number_of_ids;
  sctp_assoc_t gaids_assoc_id[];
};
```

The caller must provide a large enough buffer to hold all association identifiers. If the buffer is too small, an error must be returned. The user can use the SCTP_GET_ASSOC_NUMBER socket option to get an idea of how large the buffer has to be. gaids_number_of_ids gives the number of elements in the array gaids_assoc_id. Note also that some or all of sctp_assoc_t returned in the array may become invalid by the time the caller gets back the result.

For a one-to-one style socket, this socket option results in an error.

8.3. Write-Only Options

The options defined in this subsection are write-only. Using this option in a getsockopt() or sctp_opt_info() call will result in an error indicating EOPNOTSUPP.

8.3.1. Set Peer Primary Address (SCTP_SET_PEER_PRIMARY_ADDR)

This call requests that the peer mark the enclosed address as the association primary (see [RFC5061]). The enclosed address must be one of the association's locally bound addresses.

The following structure is used to make a set peer primary request:

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

sspp_assoc_id: This parameter is ignored for one-to-one style sockets. For one-to-many style sockets, it identifies the association for this request. Note that the predefined constants are not allowed for this option.

sspp_addr: The address to set as primary.

8.3.2. Add a Chunk That Must Be Authenticated (SCTP_AUTH_CHUNK)

This set option adds a chunk type that the user is requesting to be received only in an authenticated way. Changes to the list of chunks will only affect future associations on the socket.

The following structure is used to add a chunk:

```
struct sctp_authchunk {
  uint8_t sauth_chunk;
};
```

sauth_chunk: This parameter contains a chunk type that the user is requesting to be authenticated.

The chunk types for INIT, INIT-ACK, SHUTDOWN-COMPLETE, and AUTH chunks must not be used. If they are used, an error must be returned. The usage of this option enables SCTP AUTH in cases where it is not required by other means (for example, the use of dynamic address reconfiguration).

8.3.3. Set a Shared Key (SCTP_AUTH_KEY)

This option will set a shared secret key that is used to build an association shared key.

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

```
struct sctp_authkey {
  sctp_assoc_t sca_assoc_id;
  uint16_t sca_keynumber;
  uint16_t sca_keylength;
  uint8_t  sca_key[];
};
```

sca_assoc_id: This parameter indicates on what association the shared key is being set. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can be used. For one-to-one style sockets, this parameter is ignored. Note, however, that on one-to-one style sockets, this option will set a key on the association if the socket is connected; otherwise, this option will set a key on the endpoint.

sca_keynumber: This parameter is the shared key identifier by which the application will refer to this shared key. If a key of the specified index already exists, then this new key will replace the old existing key. Note that shared key identifier '0' defaults to a null key.

sca_keylength: This parameter is the length of the array sca_key.

sca_key: This parameter contains an array of bytes that is to be used by the endpoint (or association) as the shared secret key. Note that if the length of this field is zero, a null key is set.

8.3.4. Deactivate a Shared Key (SCTP_AUTH_DEACTIVATE_KEY)

This set option indicates that the application will no longer send user messages using the indicated key identifier.

```
struct sctp_authkeyid {
  sctp_assoc_t scact_assoc_id;
  uint16_t scact_keynumber;
};
```

scact_assoc_id: This parameter indicates from which association the shared key identifier is being deleted. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can be used. For one-to-one style sockets, this parameter is ignored. Note, however, that this option will deactivate the key from the association if the socket is connected; otherwise, this option will deactivate the key from the endpoint.

scact_keynumber: This parameter is the shared key identifier that the application is requesting to be deactivated. The key identifier must correspond to an existing shared key. Note that if this parameter is zero, use of the null key identifier '0' is deactivated on the endpoint and/or association.

The currently active key cannot be deactivated.

8.3.5. Delete a Shared Key (SCTP_AUTH_DELETE_KEY)

This set option will delete an SCTP association's shared secret key that has been deactivated.

```
struct sctp_authkeyid {
  sctp_assoc_t scact_assoc_id;
  uint16_t scact_keynumber;
};
```

scact_assoc_id: This parameter indicates from which association the shared key identifier is being deleted. The special SCTP_{FUTURE|CURRENT|ALL}_ASSOC can be used. For one-to-one style sockets, this parameter is ignored. Note, however, that this option will delete the key from the association if the socket is connected; otherwise, this option will delete the key from the endpoint.

scact_keynumber: This parameter is the shared key identifier that the application is requesting to be deleted. The key identifier must correspond to an existing shared key and must not be in use for any packet being sent by the SCTP implementation. This means, in particular, that it must be deactivated first. Note that if this parameter is zero, use of the null key identifier '0' is deleted from the endpoint and/or association.

Only deactivated keys that are no longer used by an association can be deleted.

9. New Functions

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

9.1. sctp_bindx()

This function allows the user to bind a specific subset of addresses or, if the SCTP extension described in [RFC5061] is supported, add or delete specific addresses.

The function prototype 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 for an IPv4 address, or as IN6ADDR_ANY_INIT or in6addr_any for an IPv6 address; 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 is used. For an IPv4 socket, an array of sockaddr_in is used. The caller specifies the number of addresses in the array with addrcnt. Note that the wildcard addresses cannot be used in combination with non-wildcard addresses on a socket 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 socket (i.e., endpoint), and SCTP_BINDX_REM_ADDR directs SCTP to remove the given addresses from the socket. The two flags are mutually exclusive; if both are given, sctp_bindx() will fail with EINVAL. A caller may not remove all addresses from a socket; 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, an application can use sctp_bindx(SCTP_BINDX_REM_ADDR) to remove some addresses with which a listening socket is associated, so that no new association accepted will be associated with these addresses. If the

endpoint supports dynamic address reconfiguration, an SCTP_BINDX_REM_ADDR or SCTP_BINDX_ADD_ADDR may cause an endpoint to send the appropriate message to its peers to change the peers' address lists.

Adding and removing addresses from established associations is an optional functionality. Implementations that do not support this functionality should return -1 and set errno to EOPNOTSUPP.

sctp_bindx() can be called on an already bound socket or on an unbound socket. If the socket is unbound and the first port number in the addrs parameter is zero, the kernel will choose a port number. All port numbers after the first one being 0 must also be zero. If the first port number is not zero, the following port numbers must be zero or have the same value as the first one. For an already bound socket, all port numbers provided must be the bound one or 0.

sctp_bindx() is an atomic operation. Therefore, the binding will either succeed on all addresses or fail on all addresses. If multiple addresses are provided and the sctp_bindx() call fails, there is no indication of which address is responsible for the failure. The only way to identify the specific error indication is to call sctp_bindx() sequentially with only one address per call.

9.2. sctp_peeloff()

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 these associations carrying high-volume data traffic into their own separate socket descriptors.

The application uses the sctp_peeloff() call to branch off an association into a separate socket. (Note that the semantics are somewhat changed from the traditional one-to-one style accept() call.) Note also 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 function prototype is

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

and the arguments are

sd: The original one-to-many style socket descriptor returned from the socket() system call (see Section 3.1.1).

assoc_id: The specified identifier of the association that is to be branched off to a separate file descriptor. (Note that 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.)

The function returns a non-negative file descriptor representing the branched-off association, or -1 if an error occurred. The variable errno is then set appropriately.

9.3. sctp_getpaddrs()

sctp_getpaddrs() returns all peer addresses in an association.

The function prototype is

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

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_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, with IPv4 addresses returned according to the SCTP_I_WANT_MAPPED_V4_ADDR option setting.

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.

9.4. sctp_freepaddrs()

sctp_freepaddrs() frees all resources allocated by sctp_getpaddrs().

The function prototype is

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

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

9.5. sctp_getladdrs()

sctp_getladdrs() returns all locally bound addresses on a socket.

The function prototype is

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

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, with IPv4 addresses returned according to the SCTP_I_WANT_MAPPED_V4_ADDR option setting.

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.

9.6. sctp_freeladdrs()

sctp_freeladdrs() frees all resources allocated by sctp_getladdrs().

The function prototype is

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

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

9.7. sctp_sendmsg() - DEPRECATED

This function is deprecated; sctp_sendv() (see Section 9.12) should be used instead.

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

The function prototype 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);
```

and the arguments are

sd: The socket descriptor.

msg: The message to be sent.

len: The length of the message.

to: The destination address of the message.

tolen: The length of the destination address.

ppid: The same as sinfo_ppid (see Section 5.3.2).

flags: The same as sinfo_flags (see Section 5.3.2).

stream_no: The same as sinfo_stream (see Section 5.3.2).

timetolive: The same as sinfo_timetolive (see Section 5.3.2).

context: The same as sinfo_context (see Section 5.3.2).

The call returns the number of characters sent, or -1 if an error occurred. The variable errno is then set appropriately.

Sending a message using sctp_sendmsg() is atomic (unless explicit EOR marking is enabled on the socket specified by sd).

Using sctp_sendmsg() on a non-connected one-to-one style socket for implicit connection setup may or may not work, depending on the SCTP implementation.

9.8. sctp_recvmsg() - DEPRECATED

This function is deprecated; sctp_recvv() (see Section 9.13) should be used instead.

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_event with the SCTP_EVENTS option. Note that the setting of the SCTP_USE_EXT_RCVINFO will affect this function as well, causing the sctp_sndrcvinfo information to be extended.

The function prototype 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);
```

and the arguments are

sd: The socket descriptor.

msg: The message buffer to be filled.

len: The length of the message buffer.

from: A pointer to an address to be filled with the address of the sender of this message.

fromlen: An in/out parameter describing the from length.

sinfo: A pointer to an sctp_sndrcvinfo structure to be filled upon receipt of the message.

msg_flags: A pointer to an integer to be filled with any message flags (e.g., MSG_NOTIFICATION). Note that this field is an in-out field. Options for the receive may also be passed into the value (e.g., MSG_PEEK). On return from the call, the msg_flags value will be different than what was sent in to the call. If implemented via a recvmsg() call, the msg_flags parameter should only contain the value of the flags from the recvmsg() call.

The call returns the number of bytes received, or -1 if an error occurred. The variable errno is then set appropriately.

9.9. 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 dependent. This function only specifies that the stack will try to make use of all of 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.

The function prototype is

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

and the arguments are

sd: The socket descriptor.

addrs: An array of addresses.

addrcnt: The number of addresses in the array.

id: An output parameter that, if passed in as non-NULL, will return the association identifier for the newly created association (if successful).

The call returns 0 on success or -1 if an error occurred. The variable errno is then set appropriately.

9.10. sctp_send() - DEPRECATED

This function is deprecated; sctp_sendv() should be used instead.

An implementation may provide another alternative function or system call to assist an application with the sending of data without the use of the cmsghdr structures.

The function prototype is

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

and the arguments are

sd: The socket descriptor.

msg: The message to be sent.

len: The length of the message.

sinfo: A pointer to an sctp_sndrcvinfo structure used as described in Section 5.3.2 for a sendmsg() call.

flags: The same flags as used by the sendmsg() call flags (e.g., MSG_DONTROUTE).

The call returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

This function call may also be used to terminate an association using an association identifier by setting the sinfo.sinfo_flags to SCTP_EOF and the sinfo.sinfo_assoc_id to the association that needs to be terminated. In such a case, len can be zero.

Using sctp_send() on a non-connected one-to-one style socket for implicit connection setup may or may not work, depending on the SCTP implementation.

Sending a message using sctp_send() is atomic unless explicit EOR marking is enabled on the socket specified by sd.

9.11. sctp_sendx() - DEPRECATED

This function is deprecated; sctp_sendv() should be used instead.

An implementation may provide another alternative function or system call to assist an application with the sending of data without the use of the cmsghdr structure, and to provide 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 9.9).

The function prototype is

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

and the arguments are

sd: The socket descriptor.

msg: The message to be sent.

len: The length of the message.

addrs: An array of addresses.

addrcnt: The number of addresses in the array.

sinfo: A pointer to an sctp_sndrcvinfo structure used as described in Section 5.3.2 for a sendmsg() call.

flags: The same flags as used by the sendmsg() call flags (e.g., MSG_DONTROUTE).

The call returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

Note that in the case of implicit connection setup, on return from this call, the sinfo_assoc_id field of the sinfo structure will contain the new association identifier.

This function call may also be used to terminate an association using an association identifier by setting the sinfo.sinfo_flags to SCTP_EOF and the sinfo.sinfo_assoc_id to the association that needs to be terminated. In such a case, len would be zero.

Sending a message using sctp_sendx() is atomic unless explicit EOR marking is enabled on the socket specified by sd.

Using sctp_sendx() on a non-connected one-to-one style socket for implicit connection setup may or may not work, depending on the SCTP implementation.

9.12. sctp_sendv()

The function prototype is

```
ssize_t sctp_sendv(int sd,
                   const struct iovec *iov,
                   int iovcnt,
                   struct sockaddr *addrs,
                   int addrcnt,
                   void *info,
                   socklen_t infolen,
                   unsigned int infotype,
                   int flags);
```

The function sctp_sendv() provides an extensible way for an application to communicate different send attributes to the SCTP stack when sending a message. An implementation may provide sctp_sendv() as a library function or a system call.

This document defines three types of attributes that can be used to describe a message to be sent. They are struct sctp_sndinfo (Section 5.3.4), struct sctp_prinfo (Section 5.3.7), and struct sctp_authinfo (Section 5.3.8). The following structure, sctp_sendv_spa, is defined to be used when more than one of the above attributes are needed to describe a message to be sent.

```
struct sctp_sendv_spa {
  uint32_t sendv_flags;
  struct sctp_sndinfo sendv_sndinfo;
  struct sctp_prinfo sendv_prinfo;
  struct sctp_authinfo sendv_authinfo;
};
```

The sendv_flags field holds a bitwise OR of SCTP_SEND_SNDINFO_VALID, SCTP_SEND_PRINFO_VALID, and SCTP_SEND_AUTHINFO_VALID indicating if the sendv_sndinfo/sendv_prinfo/sendv_authinfo fields contain valid information.

In future, when new send attributes are needed, new structures can be defined. But those new structures do not need to be based on any of the above defined structures.

The function takes the following arguments:

sd: The socket descriptor.

iov: The gather buffer. The data in the buffer is treated as a single user message.

iovcnt: The number of elements in iov.

addrs: An array of addresses to be used to set up an association or a single address to be used to send the message. NULL is passed in if the caller neither wants to set up an association nor wants to send the message to a specific address.

addrcnt: The number of addresses in the addrs array.

info: A pointer to the buffer containing the attribute associated with the message to be sent. The type is indicated by the info_type parameter.

infolen: The length of info, in bytes.

infotype: Identifies the type of the information provided in info. The current defined values are as follows:

SCTP_SENDV_NOINFO: No information is provided. The parameter info is a NULL pointer, and infolen is 0.

SCTP_SENDV_SNDINFO: The parameter info is pointing to a struct sctp_sndinfo.

SCTP_SENDV_PRINFO: The parameter info is pointing to a struct sctp_prinfo.

SCTP_SENDV_AUTHINFO: The parameter info is pointing to a struct sctp_authinfo.

SCTP_SENDV_SPA: The parameter info is pointing to a struct sctp_sendv_spa.

flags: The same flags as used by the sendmsg() call flags (e.g., MSG_DONTROUTE).

The call returns the number of bytes sent, or -1 if an error occurred. The variable errno is then set appropriately.

A note on the one-to-many style socket: The struct sctp_sndinfo attribute must always be used in order to specify the association on which the message is to be sent. The only case where it is not needed is when this call is used to set up a new association.

The caller provides a list of addresses in the addrs parameter to set up an association. This function will behave like calling sctp_connectx() (see Section 9.9), first using the list of addresses and then calling sendmsg() with the given message and attributes. For a one-to-many style socket, if the struct sctp_sndinfo attribute is provided, the snd_assoc_id field must be 0. When this function returns, the snd_assoc_id field will contain the association identifier of the newly established association. Note that the struct sctp_sndinfo attribute is not required to set up an association for a one-to-many style socket. If this attribute is not provided, the caller can enable the SCTP_ASSOC_CHANGE notification and use the SCTP_COMM_UP message to find out the association identifier.

If the caller wants to send the message to a specific peer address (hence overriding the primary address), it can provide the specific address in the addrs parameter and provide a struct sctp_sndinfo attribute with the field snd_flags set to SCTP_ADDR_OVER.

This function call may also be used to terminate an association. The caller provides an sctp_sndinfo attribute with the snd_flags set to SCTP_EOF. In this case, len would be zero.

Sending a message using sctp_sendv() is atomic unless explicit EOR marking is enabled on the socket specified by sd.

9.13. sctp_recvv()

The function prototype is

```
ssize_t sctp_recvv(int sd,
                   const struct iovec *iov,
                   int iovlen,
                   struct sockaddr *from,
                   socklen_t *fromlen,
                   void *info,
                   socklen_t *infolen,
                   unsigned int *infotype,
                   int *flags);
```

The function sctp_recvv() provides an extensible way for the SCTP stack to pass up different SCTP attributes associated with a received message to an application. An implementation may provide sctp_recvv() as a library function or as a system call.

This document defines two types of attributes that can be returned by this call: the attribute of the received message and the attribute of the next message in the receive buffer. The caller enables the SCTP_RECVRCVINFO and SCTP_RECVNXTINFO socket options, respectively, to receive these attributes. Attributes of the received message are returned in struct sctp_rcvinfo (Section 5.3.5), and attributes of the next message are returned in struct sctp_nxtinfo (Section 5.3.6). If both options are enabled, both attributes are returned using the following structure.

```
struct sctp_recvv_rn {
  struct sctp_rcvinfo recvv_rcvinfo;
  struct sctp_nxtinfo recvv_nxtinfo;
};
```

In future, new structures can be defined to hold new types of attributes. The new structures do not need to be based on struct sctp_recvv_rn or struct sctp_rcvinfo.

This function takes the following arguments:

sd: The socket descriptor.

iov: The scatter buffer. Only one user message is returned in this buffer.

iovlen: The number of elements in iov.

from: A pointer to an address to be filled with the sender of the received message's address.

fromlen: An in/out parameter describing the from length.

info: A pointer to the buffer to hold the attributes of the received message. The structure type of info is determined by the info_type parameter.

infolen: An in/out parameter describing the size of the info buffer.

infotype: On return, *info_type is set to the type of the info buffer. The current defined values are as follows:

SCTP_RECVV_NOINFO: If both SCTP_RECVRCVINFO and SCTP_RECVNXTINFO options are not enabled, no attribute will be returned. If only the SCTP_RECVNXTINFO option is enabled but there is no next message in the buffer, no attribute will be returned. In these cases, *info_type will be set to SCTP_RECVV_NOINFO.

SCTP_RECVV_RCVINFO: The type of info is struct sctp_rcvinfo, and the attribute relates to the received message.

SCTP_RECVV_NXTINFO: The type of info is struct sctp_nxtinfo, and the attribute relates to the next message in the receive buffer. This is the case when only the SCTP_RECVNXTINFO option is enabled and there is a next message in the buffer.

SCTP_RECVV_RN: The type of info is struct sctp_recvv_rn. The recvv_rcvinfo field is the attribute of the received message, and the recvv_nxtinfo field is the attribute of the next message in the buffer. This is the case when both SCTP_RECVRCVINFO and SCTP_RECVNXTINFO options are enabled and there is a next message in the receive buffer.

flags: A pointer to an integer to be filled with any message flags (e.g., MSG_NOTIFICATION). Note that this field is an in/out parameter. Options for the receive may also be passed into the value (e.g., MSG_PEEK). On return from the call, the flags value will be different than what was sent in to the call. If implemented via a recvmsg() call, the flags should only contain the value of the flags from the recvmsg() call when calling sctp_recvv(), and on return it has the value from msg_flags.

The call returns the number of bytes received, or -1 if an error occurred. The variable errno is then set appropriately.

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, unprivileged users should not be able to set protocol parameters that could result in the congestion control algorithm being more aggressive than permitted on the public Internet. These parameters are as follows:

- struct sctp_rtoinfo

If an unprivileged user inherits a one-to-many style socket with open associations on a privileged port, accepting new associations might be permitted, but opening new associations should not be permitted. This could be relevant for the r* family (rsh, rlogin, rwho, ...) of protocols.

Applications using the one-to-many style sockets and using the interleave level (if 0) are subject to denial-of-service attacks, as described in Section 8.1.20.

Applications needing transport layer security can use Datagram Transport Layer Security/SCTP (DTLS/SCTP) as specified in [RFC6083]. This can be implemented using the sockets API described in this document.

11. Acknowledgments

Special acknowledgment is given to Ken Fujita, Jonathan Woods, Qiaobing Xie, and La Monte Yarroll, who helped extensively in the early formation of this document.

The authors also wish to thank Kavitha Baratakke, Mike Bartlett, Martin Becke, Jon Berger, Mark Butler, Thomas Dreibholz, Andreas Fink, Scott Kimble, Jonathan Leighton, Renee Revis, Irene Ruengeler, Dan Wing, 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

12.1. Normative References

[IEEE-1003.1-2008]
Institute of Electrical and Electronics Engineers, "Information Technology - Portable Operating System Interface (POSIX)", IEEE Standard 1003.1, 2008.

[RFC3493] Gilligan, R., Thomson, S., Bound, J., McCann, J., and W. Stevens, "Basic Socket Interface Extensions for IPv6", RFC 3493, February 2003.

[RFC3542] Stevens, W., Thomas, M., Nordmark, E., and T. Jinmei, "Advanced Sockets Application Program Interface (API) for IPv6", RFC 3542, May 2003.

[RFC3758] Stewart, R., Ramalho, M., Xie, Q., Tuexen, M., and P. Conrad, "Stream Control Transmission Protocol (SCTP) Partial Reliability Extension", RFC 3758, May 2004.

[RFC4895] Tuexen, M., Stewart, R., Lei, P., and E. Rescorla, "Authenticated Chunks for the Stream Control Transmission Protocol (SCTP)", RFC 4895, August 2007.

[RFC4960] Stewart, R., Ed., "Stream Control Transmission Protocol", RFC 4960, September 2007.

[RFC5061] Stewart, R., Xie, Q., Tuexen, M., Maruyama, S., and M. Kozuka, "Stream Control Transmission Protocol (SCTP) Dynamic Address Reconfiguration", RFC 5061, September 2007.

12.2. Informative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1644] Braden, R., "T/TCP -- TCP Extensions for Transactions Functional Specification", RFC 1644, July 1994.

[RFC6083] Tuexen, M., Seggelmann, R., and E. Rescorla, "Datagram Transport Layer Security (DTLS) for Stream Control Transmission Protocol (SCTP)", RFC 6083, January 2011.

[RFC6247] Eggert, L., "Moving the Undeployed TCP Extensions RFC 1072, RFC 1106, RFC 1110, RFC 1145, RFC 1146, RFC 1379, RFC 1644, and RFC 1693 to Historic Status", RFC 6247, May 2011.

Appendix A. Example Using One-to-One Style Sockets

The following code is an implementation of a simple client that sends a number of messages marked for unordered delivery to an echo server making use of all outgoing streams. The example shows how to use some features of one-to-one style IPv4 SCTP sockets, including

- Creating and connecting an SCTP socket.
- Making a request to negotiate a number of outgoing streams.
- Determining the negotiated number of outgoing streams.
- Setting an adaptation layer indication.
- Sending messages with a given payload protocol identifier on a particular stream using sctp_sendv().

<CODE BEGINS>

```
/*

   Copyright (c) 2011 IETF Trust and the persons identified
   as authors of the code.  All rights reserved.

   Redistribution and use in source and binary forms, with
   or without modification, is permitted pursuant to, and subject
   to the license terms contained in, the Simplified BSD License
   set forth in Section 4.c of the IETF Trust's Legal Provisions
   Relating to IETF Documents (http://trustee.ietf.org/license-info).

*/

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

#define PORT 9
#define ADDR "127.0.0.1"
#define SIZE_OF_MESSAGE 1000
#define NUMBER_OF_MESSAGES 10
#define PPID 1234
```

```
int
main(void) {
  unsigned int i;
  int sd;
  struct sockaddr_in addr;
  char buffer[SIZE_OF_MESSAGE];
  struct iovec iov;
  struct sctp_status status;
  struct sctp_initmsg init;
  struct sctp_sndinfo info;
  struct sctp_setadaptation ind;
  socklen_t opt_len;

  /* Create a one-to-one style SCTP socket. */
  if ((sd = socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP)) < 0) {
    perror("socket");
    exit(1);
  }

  /* Prepare for requesting 2048 outgoing streams. */
  memset(&init, 0, sizeof(init));
  init.sinit_num_ostreams = 2048;
  if (setsockopt(sd, IPPROTO_SCTP, SCTP_INITMSG,
                 &init, (socklen_t)sizeof(init)) < 0) {
    perror("setsockopt");
    exit(1);
  }

  ind.ssb_adaptation_ind  = 0x01020304;
  if (setsockopt(sd, IPPROTO_SCTP, SCTP_ADAPTATION_LAYER,
                 &ind, (socklen_t)sizeof(ind)) < 0) {
    perror("setsockopt");
    exit(1);
  }

  /* Connect to the discard server. */
  memset(&addr, 0, sizeof(addr));
#ifdef HAVE_SIN_LEN
  addr.sin_len         = sizeof(struct sockaddr_in);
#endif
  addr.sin_family      = AF_INET;
  addr.sin_port        = htons(PORT);
  addr.sin_addr.s_addr = inet_addr(ADDR);
```

```
    if (connect(sd,
                (const struct sockaddr *)&addr,
                sizeof(struct sockaddr_in)) < 0) {
      perror("connect");
      exit(1);
    }

    /* Get the actual number of outgoing streams. */
    memset(&status, 0, sizeof(status));
    opt_len = (socklen_t)sizeof(status);
    if (getsockopt(sd, IPPROTO_SCTP, SCTP_STATUS,
                   &status, &opt_len) < 0) {
      perror("getsockopt");
      exit(1);
    }

    memset(&info, 0, sizeof(info));
    info.snd_ppid = htonl(PPID);
    info.snd_flags = SCTP_UNORDERED;
    memset(buffer, 'A', SIZE_OF_MESSAGE);
    iov.iov_base = buffer;
    iov.iov_len = SIZE_OF_MESSAGE;
    for (i = 0; i <  NUMBER_OF_MESSAGES; i++) {
      info.snd_sid = i % status.sstat_outstrms;
      if (sctp_sendv(sd,
                     (const struct iovec *)&iov, 1,
                     NULL, 0,
                     &info, sizeof(info), SCTP_SENDV_SNDINFO,
                     0) < 0) {
        perror("sctp_sendv");
        exit(1);
      }
    }

    if (close(sd) < 0) {
      perror("close");
      exit(1);
    }
    return(0);
  }
  <CODE ENDS>
```

Appendix B. Example Using One-to-Many Style Sockets

The following code is a simple implementation of a discard server over SCTP. The example shows how to use some features of one-to-many style IPv6 SCTP sockets, including

- Opening and binding of a socket.
- Enabling notifications.
- Handling notifications.
- Configuring the auto-close timer.
- Using sctp_recvv() to receive messages.

Please note that this server can be used in combination with the client described in Appendix A.

```
<CODE BEGINS>
/*

   Copyright (c) 2011 IETF Trust and the persons identified
   as authors of the code.  All rights reserved.

   Redistribution and use in source and binary forms, with
   or without modification, is permitted pursuant to, and subject
   to the license terms contained in, the Simplified BSD License
   set forth in Section 4.c of the IETF Trust's Legal Provisions
   Relating to IETF Documents (http://trustee.ietf.org/license-info).

*/

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

#define BUFFER_SIZE (1<<16)
#define PORT 9
#define ADDR "0.0.0.0"
#define TIMEOUT 5
```

```
   static void
   print_notification(void *buf)
   {
     struct sctp_assoc_change *sac;
     struct sctp_paddr_change *spc;
     struct sctp_adaptation_event *sad;
     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("^^^ Association change: ");
       switch (sac->sac_state) {
       case SCTP_COMM_UP:
         printf("Communication up (streams (in/out)=(%u/%u)).\n",
                sac->sac_inbound_streams, sac->sac_outbound_streams);
         break;
       case SCTP_COMM_LOST:
         printf("Communication lost (error=%d).\n", sac->sac_error);
         break;
       case SCTP_RESTART:
         printf("Communication restarted (streams (in/out)=(%u/%u).\n",
                sac->sac_inbound_streams, sac->sac_outbound_streams);
         break;
       case SCTP_SHUTDOWN_COMP:
         printf("Communication completed.\n");
         break;
       case SCTP_CANT_STR_ASSOC:
         printf("Communication couldn't be started.\n");
         break;
       default:
         printf("Unknown state: %d.\n", sac->sac_state);
         break;
       }
       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("^^^ Peer Address change: %s ", ap);
      switch (spc->spc_state) {
      case SCTP_ADDR_AVAILABLE:
        printf("is available.\n");
        break;
      case SCTP_ADDR_UNREACHABLE:
        printf("is not available (error=%d).\n", spc->spc_error);
        break;
      case SCTP_ADDR_REMOVED:
        printf("was removed.\n");
        break;
      case SCTP_ADDR_ADDED:
        printf("was added.\n");
        break;
      case SCTP_ADDR_MADE_PRIM:
        printf("is primary.\n");
        break;
      default:
        printf("unknown state (%d).\n", spc->spc_state);
        break;
      }
      break;
    case SCTP_SHUTDOWN_EVENT:
      printf("^^^ Shutdown received.\n");
      break;
    case SCTP_ADAPTATION_INDICATION:
      sad = &snp->sn_adaptation_event;
      printf("^^^ Adaptation indication 0x%08x received.\n",
             sad->sai_adaptation_ind);
      break;
    default:
      printf("^^^ Unknown event of type: %u.\n",
             snp->sn_header.sn_type);
      break;
    };
  }
```

```
int
main(void) {
  int sd, flags, timeout, on;
  ssize_t n;
  unsigned int i;
  union {
    struct sockaddr sa;
    struct sockaddr_in sin;
    struct sockaddr_in6 sin6;
  } addr;
  socklen_t fromlen, infolen;
  struct sctp_rcvinfo info;
  unsigned int infotype;
  struct iovec iov;
  char buffer[BUFFER_SIZE];
  struct sctp_event event;
  uint16_t event_types[] = {SCTP_ASSOC_CHANGE,
                            SCTP_PEER_ADDR_CHANGE,
                            SCTP_SHUTDOWN_EVENT,
                            SCTP_ADAPTATION_INDICATION};

  /* Create a one-to-many style SCTP socket. */
  if ((sd = socket(AF_INET6, SOCK_SEQPACKET, IPPROTO_SCTP)) < 0) {
    perror("socket");
    exit(1);
  }

  /* Enable the events of interest. */
  memset(&event, 0, sizeof(event));
  event.se_assoc_id = SCTP_FUTURE_ASSOC;
  event.se_on = 1;
  for (i = 0; i < sizeof(event_types)/sizeof(uint16_t); i++) {
    event.se_type = event_types[i];
    if (setsockopt(sd, IPPROTO_SCTP, SCTP_EVENT,
                   &event, sizeof(event)) < 0) {
      perror("setsockopt");
      exit(1);
    }
  }

  /* Configure auto-close timer. */
  timeout = TIMEOUT;
  if (setsockopt(sd, IPPROTO_SCTP, SCTP_AUTOCLOSE,
                 &timeout, sizeof(timeout)) < 0) {
    perror("setsockopt SCTP_AUTOCLOSE");
    exit(1);
  }
```

```
    /* Enable delivery of SCTP_RCVINFO. */
    on = 1;
    if (setsockopt(sd, IPPROTO_SCTP, SCTP_RECVRCVINFO,
                   &on, sizeof(on)) < 0) {
      perror("setsockopt SCTP_RECVRCVINFO");
      exit(1);
    }

    /* Bind the socket to all local addresses. */
    memset(&addr, 0, sizeof(addr));
  #ifdef HAVE_SIN6_LEN
    addr.sin6.sin6_len         = sizeof(addr.sin6);
  #endif
    addr.sin6.sin6_family      = AF_INET6;
    addr.sin6.sin6_port        = htons(PORT);
    addr.sin6.sin6_addr        = in6addr_any;
    if (bind(sd, &addr.sa, sizeof(addr.sin6)) < 0) {
      perror("bind");
      exit(1);
    }
    /* Enable accepting associations. */
    if (listen(sd, 1) < 0) {
      perror("listen");
      exit(1);
    }

    for (;;) {
      flags = 0;
      memset(&addr, 0, sizeof(addr));
      fromlen = (socklen_t)sizeof(addr);
      memset(&info, 0, sizeof(info));
      infolen = (socklen_t)sizeof(info);
      infotype = 0;
      iov.iov_base = buffer;
      iov.iov_len = BUFFER_SIZE;

      n = sctp_recvv(sd, &iov, 1,
                     &addr.sa, &fromlen,
                     &info, &infolen, &infotype,
                     &flags);

      if (flags & MSG_NOTIFICATION) {
        print_notification(iov.iov_base);
      } else {
        char addrbuf[INET6_ADDRSTRLEN];
        const char *ap;
        in_port_t port;
```

```
        if (addr.sa.sa_family == AF_INET) {
               ap = inet_ntop(AF_INET, &addr.sin.sin_addr,
                              addrbuf, INET6_ADDRSTRLEN);
               port = ntohs(addr.sin.sin_port);
        } else {
               ap = inet_ntop(AF_INET6, &addr.sin6.sin6_addr,
                              addrbuf, INET6_ADDRSTRLEN);
               port = ntohs(addr.sin6.sin6_port);
        }
        printf("Message received from %s:%u: len=%d",
               ap, port, (int)n);
        switch (infotype) {
        case SCTP_RECVV_RCVINFO:
          printf(", sid=%u", info.rcv_sid);
          if (info.rcv_flags & SCTP_UNORDERED) {
            printf(", unordered");
          } else {
            printf(", ssn=%u", info.rcv_ssn);
          }
          printf(", tsn=%u", info.rcv_tsn);
          printf(", ppid=%u.\n", ntohl(info.rcv_ppid));
          break;
        case SCTP_RECVV_NOINFO:
        case SCTP_RECVV_NXTINFO:
        case SCTP_RECVV_RN:
          printf(".\n");
          break;
        default:
          printf(" unknown infotype.\n");
        }
      }
    }

    if (close(sd) < 0) {
      perror("close");
      exit(1);
    }

    return (0);
  }
  <CODE ENDS>
```

Authors' Addresses

Randall R. Stewart
Adara Networks
Chapin, SC 29036
USA

EMail: randall@lakerest.net

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

EMail: tuexen@fh-muenster.de

Kacheong Poon
Oracle Corporation

EMail: ka-cheong.poon@oracle.com

Peter Lei
Cisco Systems, Inc.
9501 Technology Blvd.
West Office Center
Rosemont, IL 60018
USA

EMail: peterlei@cisco.com

Vladislav Yasevich
HP
110 Spitrook Rd.
Nashua, NH 03062
USA

EMail: vladislav.yasevich@hp.com