```
    struct QOStype sc_qos;

#define SC_SIZE         512
    int     sc_cc;
    char    sc_data[SC_SIZE];
};
```

The elements of this structure are:

`sc_sd`: the session-descriptor to be used to reference this connection;

`sc_connect`: the connection identifier used by the responder (consult page 52 for a description of the `SSAPref` structure);

`sc_responding`: the responding peer's address (which is the same as the `called` parameter given to `SConnRequest`);

`sc_result`: the connection response;

`sc_requirements`: the (negotiated) session requirements;

`sc_settings`: the (negotiated) initial token settings;

`sc_please`: the tokens which the responder wants to own (if any);

`sc_isn`: the (negotiated) initial serial-number;

`sc_ssdusize`: the largest atomic SSDU size that can be used on the connection (see the note on page 45);

`sc_qos`: the quality of service on the connection (see Section 4.6.2); and,

`sc_data/sc_cc`: any initial data (and the length of that data).

If the call to `SConnRequest` is successful, and the `sc_result` element is set to `SC_ACCEPT`, then connection establishment has completed and the users of the session service now operate as symmetric peers. If the call is successful, but the `sc_result` element is not `SC_ACCEPT`, then the connection has been rejected; consult Table 3.4 to determine the reason (further information can

be found in the si parameter). Otherwise, if the call fails then the connection is not established and the SSAPabort structure of the SSAPindication discriminated union has been updated.

Normally SConnRequest returns only after a connection has succeeded or failed. This is termed a *synchronous* connection initiation. If the user desires, an *asynchronous* connection may be initiated. The routine SConnRequest is really a macro which calls the routine SAsynConnRequest with an argument indicating that a connection should be attempted synchronously.

```
int     SAsynConnRequest (ref, calling, called,
                requirements, settings, isn, data, cc,
                qos, sc, si, async)
struct SSAPref *ref;
struct SSAPaddr *calling,
                *called;
int     requirements,
        settings,
        cc,
        async;
long isn;
char   *data;
struct QOStype *qos;
struct SSAPconnect *sc;
struct SSAPindication *si;
```

The additional parameter to this procedure is:

async: whether the connection should be initiated asynchronously.

If the async parameter is non-zero, then SAsynConnRequest returns one of four values: NOTOK, which indicates that the connection request failed; DONE, which indicates that the connection request succeeded; or, either of CONNECTING_1 or CONNECTING_2, which indicates that the connection request is still in progress. In the first two cases, the usual procedures for handling return values from SConnRequest are employed (i.e., a NOTOK return from SAsynConnRequest is equivalent to a NOTOK return from SConnRequest, and, a DONE return from SAsynConnRequest is equivalent to a OK return from SConnRequest). In the final case, when either CONNECTING_1 or CONNECTING_2 is returned, only the sc_sd element of the sc parameter has been updated;

it reflects the session-descriptor to be used to reference this connection. Note that the `data` parameter is still being referenced by *libssap*(3n) and should not be tampered with until the connection attempt has been completed.

To determine when the connection attempt has been completed, the routine `xselect` (consult Section 2.4 of *Volume One*) should be used after calling `SSelectMask`. In order to determine if the connection attempt is successful, the `SAsynRetryRequest` routine is called:

```
int     SAsynRetryRequest (sd, sc, si)
int     sd;
struct SSAPconnect *sc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`sc:` a pointer to a `SSAPconnect` structure, which is updated only if the call succeeds; and,

`si:` a pointer to a `SSAPindication` structure, which is updated only if the call fails.

Again, one of three values are returned: `NOTOK`, which indicates that the connection request failed; `DONE`, which indicates that the connection request succeeded; and, `CONNECTING_1` or `CONNECTING_2` which indicates that the connection request is still in progress.

Refer to Section 4.2.3 on page 112 for information on how to make efficient use of the asynchronous connection facility.

3.4 Data Transfer

Once the connection has been established, a session-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libssap*(3n) library. Further, the last parameter is usually a pointer to a `SSAPindication` structure (as described on page 53). If a call to one of these routines fails, then the structure is updated. Consult the `SSAPabort` element of the `SSAPindication` structure. If the `sa_reason` element of the `SSAPabort` structure is associated with a fatal error, then the

connection is closed. That is, a S-P-ABORT.INDICATION event has occurred. The SC_FATAL macro can be used to determine this.

```
int     SC_FATAL (r)
int     r;
```

The most common interface error to occur is SC_OPERATION which usually indicates that either the user is lacking ownership of a session token to perform an operation, or that a session requirement required by the operation was not negotiated during connection establishment. For protocol purists, the SC_OFFICIAL macro can be used to determine if the error is an "official" error as defined by the specification, or an "unofficial" error used by the implementation.

```
int     SC_OFFICIAL (r)
int     r;
```

3.4.1 Sending Data

There are six routines which may be used to send data. A call to the SDataRequest routine is equivalent to a S-DATA.REQUEST action on the part of the user.

```
int     SDataRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

sd: the session-descriptor;

data/cc: the data to be written (and the length of that data); and,

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

If the call to SDataRequest is successful, then the data has been queued for sending to the peer. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure.

A call to the SExpdRequest routine is equivalent to a S-EXPEDITED-DATA.REQUEST action on the part of the user.

```
int     SExpdRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: the data to be written (and the length of that data, which may not exceed `SX_SIZE` octets); and,

`si`: a pointer to a `SSAPindication` structure, which is updated only if the call fails.

If the call to `SExpdRequest` is successful, then the data has been queued for expedited sending. Otherwise the `SSAPabort` element of the `si` parameter contains the reason for failure.

A call to the `STypedRequest` routine is equivalent to a S-TYPED-DATA.REQUEST action on the part of the user.

```
int     STypedRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd`: the session-descriptor;

`data/cc`: the data to be written (and the length of that data); and,

`si`: a pointer to a `SSAPindication` structure, which is updated only if the call fails.

If the call to `STypedRequest` is successful, then the data has been queued for sending to the peer. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

A call to the `SCapdRequest` routine is equivalent to a S-CAPABILITY-DATA.REQUEST action on the part of the user.

```
int     SCapdRequest (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`data/cc:` the data to be written (and the length of that data, which may not exceed `SX_CDSIZE` octets); and,

`si:` a pointer to a `SSAPindication` structure, which is updated only if the call fails.

If the call to `SCapdRequest` is successful, then the data has been queued for sending. When the S-CAPABILITY-DATA.CONFIRMATION event is received, the data has been successfully received. Otherwise the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

Upon receiving a S-CAPABILITY-DATA.INDICATION event, the user is required to generate a S-CAPABILITY-DATA.RESPONSE action using the `SCapdResponse` routine.

```
int     SCapdResponse (sd, data, cc, si)
int     sd;
char   *data;
int     cc;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`data/cc:` the data to be written (and the length of that data, which may not exceed `SX_CDASIZE` octets); and,

`si:` a pointer to a `SSAPindication` structure, which is updated only if the call fails.

If the call to SCapdResponse is successful, then the data has been queued for sending to the peer. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure.

The SWriteRequest routine is similar in nature to the SDataRequest and STypedRequest routines, but uses a different set of parameters. The invocation is:

```
int     SWriteRequest (sd, typed, uv, si)
int     sd;
int     typed;
struct udvec *uv;
int     cc;
```

The parameters to this procedure are:

sd: the session-descriptor;

typed: whether this is typed-data or not;

uv: the data to be written, described in a null-terimated array of scatter/gather elements; and,

si: a pointer to a SSAPindication structure, which is updated only if the call fails.

If the call to SWriteRequest is successful, then the data has been queued for sending to the peer. Otherwise the SSAPabort structure contained in the SSAPindication parameter si contains the reason for failure.

3.4.2 Receiving Data

There is one routine used to read data, SReadRequest, a call to which is equivalent to waiting for an event (usually an incoming data event) to occur.

```
int     SReadRequest (sd, sx, secs, si)
int     sd;
struct SSAPdata *sx;
int     secs;
struct SSAPindication  *si;
```

The parameters to this procedure are:

`sd:` the session-descriptor;

`sx:` a pointer to the `SSAPdata` structure to be given the data;

`secs:` the maximum number of seconds to wait for the data (a value of `NOTOK` indicates that the call should block indefinitely, whereas a value of `OK` indicates that the call should not block at all, e.g., a polling action); and,

`si:` a pointer to a `SSAPindication` structure, which is updated only if data is not read.

Unlike the other routines in the *libssap*(3n) library, the `SReadRequest` routine returns one of three values: `NOTOK` (on failure), `OK` (on reading data), or `DONE` (otherwise).

If the call to `SReadRequest` returns the manifest constant `OK`, then the data has been read into the `sx` parameter, which is a pointer to a `SSAPdata` structure.

```
struct SSAPdata {
    int     sx_type;
#define SX_NORMAL       0x00
#define SX_EXPEDITED    0x01
#define SX_TYPED        0x02
#define SX_CAPDIND      0x03
#define SX_CAPDCNF      0x04

    int     sx_cc;
    struct qbuf sx_qbuf;
};
```

The elements of a `SSAPdata` structure are:

`sx_type:` indicates how the data was received:

Value	Event
SX_NORMAL	S-DATA.INDICATION
SX_EXPEDITED	S-EXPEDITED-DATA.INDICATION
SX_TYPED	S-TYPED-DATA.INDICATION
SX_CAPDIND	S-CAPABILITY-DATA.INDICATION
SX_CAPDCNF	S-CAPABILITY-DATA.CONFIRMATION

`sx_cc`: the total number of octets that was read; and,

`sx_qbuf`: the data that was read in a buffer-queue form (see page 119 for a description of this structure).

Note that the data contained in the structure was allocated via *malloc*(3), and should be released by using the `SXFREE` macro when no longer referenced. The `SXFREE` macro, behaves as if it was defined as:

```
void    SXFREE (sx)
struct SSAPdata *sx;
```

The macro frees only the data allocated by `SReadRequest`, and not the `SSAPdata` structure itself. Further, `SXFREE` should be called only if the call to the `SReadRequest` routine returned `OK`.

> **NOTE:** Because the `SSAPdata` structure contains a `qbuf` element, care must be taken in initializing and copying variables of this type. The routines in *libssap*(3n) library will correctly initialize these structures when given as parameters. But, users who otherwise manipulate these structures should take great care.

Otherwise if the call to `SReadRequest` returns the manifest constant `NOTOK`, then the `SSAPabort` structure contained in the `SSAPindication` parameter `si` contains the reason for failure.

If the call to `SReadRequest` returns the manifest constant `DONE`, then some event other than data arrival has occurred. This event is encoded in the `si` parameter, depending on the value of the `si_type` element. When `SReadRequest` returns `DONE`, the `si_type` element may be set to one of five values:

Value	Event
`SI_TOKEN`	Token management
`SI_SYNC`	Synchronization management
`SI_ACTIVITY`	Activity management
`SI_REPORT`	Exception reporting
`SI_FINISH`	Connection release

Token Indications

When an event associated with token management occurs, the `si_type` field of the `si` parameter contains the value `SI_TOKEN`, and a `SSAPtoken` structure is contained inside the `si` parameter.

```
struct SSAPtoken {
    int     st_type;
#define ST_GIVE         0x00
#define ST_PLEASE       0x01
#define ST_CONTROL      0x02

    u_char  st_tokens;
    u_char  st_owned;

#define ST_SIZE         512
    int     st_cc;
    char    st_data[ST_SIZE];
};
```

The elements of a `SSAPtoken` structure are:

`st_type:` defines the token management indication which occurred:

Value	Event
ST_GIVE	S-TOKEN-GIVE.INDICATION
ST_PLEASE	S-TOKEN-PLEASE.INDICATION
ST_CONTROL	S-GIVE-CONTROL.INDICATION

`st_tokens:` the session tokens requested or given;

`st_owned:` all of the session tokens currently owned by the user; and,

`st_base/st_cc:` associated data (and the length of that data) if tokens are requested.

It is entirely at the discretion of the user what actions are to be taken when an indication event associated with token management occurs.