```
    int    qb_len;
    char  *qb_data;

    char   qb_base[1];
};
```

The elements of a `qbuf` structure are:

`qb_forw/qb_back:` forward and back pointers;

`qb_data/qb_len:` the user data (and the length of that data); and,

`qb_base:` the extensible array containing the data.

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

```
void    TXFREE (tx)
struct TSAPdata *tx;
```

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

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

Otherwise if the call to `TReadRequest` did not succeed, the `td` parameter indicates the reason for failure.

4.3.3 Asynchronous Event Handling

The data transfer events discussed thus far have been synchronous in nature. Some users of the transport service may wish an asynchronous interface.

The TSetIndications routine is used to change the service associated with a transport-descriptor to or from an asynchronous interface.

```
int     TSetIndications (sd, data, disc, td)
int     sd;
int   (*data) (),
      (*disc) ();
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

sd: the transport-descriptor;

data: the address of an event-handler routine to be invoked when data has arrived (either a T-DATA.INDICATION or T-EXPEDITED-DATA.INDICATION event occurs);

disc: the address of an event-handler routine to be invoked when the connection has been closed (a T-DISCONNECT.INDICATION event occurs); and,

td: a pointer to a TSAPdisconnect structure, which is updated only if the call fails.

If the service is to be made asynchronous, then both data and disc are specified; otherwise, if the service is to be made synchronous, neither should be specified (use the manifest constant NULLIFP). The most likely reason for the call failing is DR_WAITING, which indicates that an event is waiting for the user.

When an event-handler is invoked, future invocations of the event-hander are blocked until it returns. The return value of the event-handler is ignored. Further, during the execution of a synchronous call to the library, the event-handler will be blocked from being invoked.

When an event associated with data arriving occurs, the event-handler routine is invoked with two parameters:

```
(*data) (sd, tx);
int     sd;
struct TSAPdata *tx;
```

The parameters are:

`sd:` the transport-descriptor; and,

`tx:` a pointer to a `TSAPdata` structure containing the data.

Note that the data contained in the structure was allocated via *malloc*(3), and should be released with the `TXFREE` macro (described on page 119) when no longer needed.

Similarly, when an event associated with connection release occurs, the event-handler is also invoked with two parameters:

```
(*disc) (sd, td);
int     sd;
struct TSAPdisconnect *td;
```

The parameters are

`sd:` the transport-descriptor; and,

`td:` a pointer to a `TSAPdisconnect` structure indicating why the connection was released.

Note that the transport-descriptor is no longer valid at the instant the call is made.

> **NOTE:** The *libtsap*(3n) library uses the SIGEMT signal to provide these services. Programs using asynchronous transport-descriptors should NOT use SIGEMT for other purposes.

4.3.4 Synchronous Event Multiplexing

A user of the transport service may wish to manage multiple transport-descriptors simultaneously; the routine `TSelectMask` is provided for this purpose. This routine updates a file-descriptor mask and associated counter for use with `xselect`.

```
int     TSelectMask (sd, mask, nfds, td)
int     sd;
fd_set *mask,
int    *nfds;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd:` the transport-descriptor;

`mask:` a pointer to a file-descriptor mask meaningful to `xselect`;

`nfds:` a pointer to an integer-valued location meaningful to `xselect`; and,

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

If the call is successful, then the `mask` and `nfds` parameters can be used as arguments to `xselect`. The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

If `xselect` indicates that the transport-descriptor is ready for reading, `TReadRequest` should be called with the `secs` parameter equal to `OK`. If the network activity does not constitute an entire event for the user, then `TReadRequest` will return `NOTOK` with error code `DR_TIMER`.

In addition, the routine `TSelectOctets` is provided to return an estimate of how many octets might be returned by the next call to `TReadRequest`:

```
int     TSelectOctets (sd, nbytes, td)
int     sd;
long   *nbytes;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd:` the transport-descriptor;

`nbytes:` a pointer to a longword location that, on success, will be updated to contain the number of octets that might be returned; and,

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

4.4 Connection Release

The `TDiscRequest` routine is used to release a connection. Note, that the TSAP makes no guarantee that any queued data will be received before the connection closes.

```
int     TDiscRequest (sd, data, cc, td)
int     sd;
char   *data;
int     cc;
struct TSAPdisconnect  *td;
```

The parameters to the procedure are described on page 110.

4.5 State Saving and Restoration

Some users of the transport service, and in particular the session provider, require the ability to *execve*(2) another process image and have that process use the transport connection. Since the *libtsap*(3n) library is not kernel-resident, special provisions are necessary to support this behavior.

4.5.1 Saving the State

The routine `TSaveState` is used to record the state of the TSAP for a given transport-descriptor.

```
int     TSaveState (sd, vec, td)
int     sd;
char  **vec;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd:` the transport-descriptor;

`vec:` a pointer to the first free slot in the argument vector for *execve*(2); and,

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

If the call succeeds, then an extra *magic argument* has been placed in the argument vector. The most likely reason for the call failing is `DR_WAITING`, which indicates that an event is waiting for the user.

> **NOTE:** Once a successful call to `TSaveState` is made on a transport descriptor, that descriptor may no longer be referenced until a corresponding call to `TRestoreState` is successful.

4.5.2 Restoring the State

The routine `TRestoreState` is used to re-initialize the state of the TSAP.

```
int     TRestoreState (buffer, ts, td)
char   *buffer;
struct TSAPstart  *ts;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`buffer:` the *magic argument* constructed by `TSaveState`;

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

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

If `TRestoreState` is successful, it returns information in a `TSAPstart` structure, as defined on page 107. There is one exception however: in the current implementation the `ts_cc` and `ts_data` elements are undefined on a successful return from `TRestoreState`.

4.6 Cookie Parameters

There are two *cookie* parameters: network addresses and quality of service.

4.6.1 Network Addresses

Network addresses can vary greatly. In this software distribution, a "unified" format has been adopted in the `NSAPaddr` structure:

```
struct NSAPaddr {
    long    na_stack;
#define NA_NSAP 0
#define NA_TCP  1
#define NA_X25  2
#define NA_BRG  3

    long    na_community;

    union {
        struct na_nsap {
#define NASIZE  64
            char    na_nsap_address[NASIZE];
            char    na_nsap_addrlen;
        }               un_na_nsap;

        struct na_tcp {
#define NSAP_DOMAINLEN  63
            char    na_tcp_domain[NSAP_DOMAINLEN + 1];
            u_short na_tcp_port;
            u_short na_tcp_tset;
#define NA_TSET_TCP    0x0001
#define NA_TSET_UDP    0x0002
        }               un_na_tcp;

        struct na_x25 {
#define NSAP_DTELEN     15
            char    na_x25_dte[NSAP_DTELEN + 1];
            char    na_x25_dtelen;

#define NPSIZE  4
            char    na_x25_pid[NPSIZE];
            char    na_x25_pidlen;

#define CUDFSIZE 16
```

```
            char    na_x25_cudf[CUDFSIZE];
            char    na_x25_cudflen;

#define FACSIZE 6
            char    na_x25_fac[FACSIZE];
            char    na_x25_faclen;
        }               un_na_x25;
    }               na_un;

#define na_address      na_un.un_na_nsap.na_nsap_address
#define na_addrlen      na_un.un_na_nsap.na_nsap_addrlen

#define na_domain       na_un.un_na_tcp.na_tcp_domain
#define na_port         na_un.un_na_tcp.na_tcp_port

#define na_dte          na_un.un_na_x25.na_x25_dte
#define na_dtelen       na_un.un_na_x25.na_x25_dtelen
#define na_pid          na_un.un_na_x25.na_x25_pid
#define na_pidlen       na_un.un_na_x25.na_x25_pidlen
#define na_cudf         na_un.un_na_x25.na_x25_cudf
#define na_cudflen      na_un.un_na_x25.na_x25_cudflen
#define na_fac          na_un.un_na_x25.na_x25_fac
#define na_faclen       na_un.un_na_x25.na_x25_faclen
};
#define NULLNA  ((struct NSAPaddr *) 0)
```

As shown, this structure is really a discriminated union (a structure with a tag element followed by a union). Based on the value of the tag (`na_stack`), a different structure is selected.

For a native OSI CO-mode transport service, the value of the tag is `NA_NSAP`, and the following elements are meaningful:

`na_address/na_addrlen:` the network address (and its length), binary-valued.

For emulation of the OSI transport service on top of the TCP, the value of the tag is `NA_TCP`, and the following elements are meaningful:

`na_domain:` the null-terminated domain name (e.g., "gonzo.twg.com") or dotted-quad (e.g., "128.99.0.17");

`na_port`: the TCP-port number offering the service (if zero, the service on port 102 is used); and,

`na_tset`: the set of IP-based transport services available at the address (if zero, the TCP service is used).

For use of a single-subnet X.25, the value of the tag is `NA_X25`, and the following elements are meaningful:

`na_dte/na_dtelen`: the X.121 address (and its length), ascii-valued, possibly null-terminated;

`na_pid/na_pidlen`: the protocol id (and its length), binary-valued;

`na_cudf/na_cudflen`: the call user data (and its length), binary-valued; and,

`na_fac/na_faclen`: the negotiated facilities proposed in the call request packet (and its length), binary-valued.

For use of a TP0-bridge between the TCP and X.25, the value of the tag is `NA_BRG`, and the elements above are meaningful.

If the value of the tag is not `NA_NSAP`, it may be useful to normalize the address into a "real" OSI address. The routine `na2norm` is used:

```
char   *na2norm (na)
struct NSAPaddr *na;
```

The parameter to this procedure is:

`na` : the network address to be normalized.

A new network address is returned from a static area which contains the normalized form.

The routine `na2str` takes a network address and returns a null-terminated ascii string suitable for viewing:

```
char   *na2str (na)
struct NSAPaddr *na;
```

The parameter to this procedure is:

`na` : the network address to be printed.

The `na_community` field is an internal number used to distinguish between different OSI communities. Consult Chapter 8 starting on page 167 for further details.

4.6.2 Quality of Service

Currently, quality of service is largely uninterpreted by the software. However, the quality of service structure contains those parameters which are supported:

```
struct QOStype {
                                /* transport QOS */
    int    qos_reliability;
#define HIGH_QUALITY   0
#define LOW_QUALITY    1

                                /* session QOS */
    int    qos_sversion;
    int    qos_extended;
    int    qos_maxtime;
};
#define NULLQOS ((struct QOStype *) 0)
```

The elements of this structure are:

`qos_reliability:` the "reliability" level of the connection, either high- or low-quality;

`qos_sversion:` the session version requested/negotiated on this connection (only applicable above the transport layer, obviously), if the manifest constant `NOTOK` is used when initiating a connection, this indicates that the highest possible version should be negotiated;

`qos_extended:` the extended control parameter for the connection, (if non-zero, extended control is used by the session layer); and,

`qos_maxtime:` after a transport connection is established, the maximum number of seconds to wait for an acknowledgement from the responding SPM (any non-positive number indicates that no time-limit is desired).