Provider-initiated Disconnect (fatal)	
DR_NORMAL	Normal disconnect by session entity
DR_REMOTE	Remote transport entity congested at connect request time
DR_CONNECT	Connection negotiation failed
DR_DUPLICATE	Duplicate source reference detected for the same pair of NSAPs
DR_MISMATCH	Mismatched references
DR_PROTOCOL	Protocol error
DR_OVERFLOW	Reference overflow
DR_REFUSED	Connect request refused on this network connection
DR_LENGTH	Header or parameter length invalid
DR_NETWORK	Network disconnect
User-initiated Disconnect (fatal)	
DR_UNKNOWN	Reason not specifed
DR_CONGEST	Congestion at TSAP
DR_SESSION	Session entity not attached to TSAP
DR_ADDRESS	Address unknown
Interface Errors (non-fatal)	
DR_PARAMETER	Invalid parameter
DR_OPERATION	Invalid operation
DR_TIMER	Timer expired
DR_WAITING	Indications waiting

Table 4.1: TSAP Failure Codes

`data/cc:` any initial data (and the length of that data, which may not exceed `TC_SIZE` octets);

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

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

If the call to `TConnResponse` is successful, then connection establishment has completed and the users of the transport service now operate as symmetric peers. Otherwise, if the call fails and the reason is not an interface error (see Table 4.1 on page 109), then the connection is closed.

If instead, the responder wishes to reject the connection, it should fire the T-DISCONNECT.REQUEST action by calling the `TDiscRequest` routine.

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

The parameters to this procedure are:

`sd:` the transport-descriptor;

`data/cc:` any disconnect data (and the length of that data, which may not exceed `TD_SIZE` octets); and,

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

After a return from this call, the responder may exit.

4.2.3 Client Initialization

A program wishing to connect to another user of transport services calls the `TConnRequest` routine, which corresponds to the T-CONNECT.REQUEST action.

```
int     TConnRequest (calling, called, expedited, data, cc,
                qos, tc, td)
struct TSAPaddr *calling,
                *called;
int     expedited,
        cc;
char   *data;
struct QOStype *qos;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`calling`: the TSAP address of the initiator; (need not be present);

`called`: the TSAP address of the responder;

`expedited`: whether expedited data is to be permitted;

`data/cc`: any initial data (and the length of that data, which may not exceed `TS_SIZE` octets);

`qos`: the quality of service on the connection (see Section 4.6.2);

`tc`: a pointer to a `TSAPconnect` 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 the call to `TConnRequest` is successful (a successful return corresponds to a T-CONNECT.CONFIRMATION event), then information is returned in the `tc` parameter, which is a pointer to a `TSAPconnect` structure.

```
struct TSAPconnect {
    int     tc_sd;

    struct TSAPaddr tc_responding;

    int     tc_expedited;
```

```
    int     tc_tsdusize;

    struct QOStype tc_qos;

#define TC_SIZE         32
    int     tc_cc;
    char    tc_data[TC_SIZE];
};
```

The elements of this structure are:

`tc_sd`: the transport-descriptor to be used to reference this connection;

`tc_responding`: the responding peer's address (which is the same as the `called` address given as a parameter to `TConnRequest`);

`tc_expedited`: whether expedited data will be supported;

`tc_tsdusize`: the largest atomic TSDU size that can be used on the connection (see the note on page 106);

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

`tc_data/tc_cc`: any initial data (and the length of that data).

If the call to `TConnRequest` is successful, then connection establishment has completed and the users of the transport service now operate as symmetric peers. Otherwise, if the call fails then the connection is not established, and the `TSAPdisconnect` structure has been updated.

Asynchronous Connections

Normally `TConnRequest` 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 `TConnRequest` is really a macro which calls the routine `TAsynConnRequest` with an argument indicating that a connection should be attempted synchronously.

```
int     TAsynConnRequest (calling, called, expedited,
                data, cc, qos, tc, td, async)
struct TSAPaddr *calling,
```

```
                *called;
int     expedited,
        cc,
        async;
char   *data;
struct QOStype *qos;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The additional parameter to this procedure is:

`async:` whether the connection should be initiated asynchronously.

If the `async` parameter is non-zero, then `TAsynConnRequest` 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 `TConnRequest` are employed (i.e., a `NOTOK` return from `TAsynConnRequest` is equivalent to a `NOTOK` return from `TConnRequest`, and, a `DONE` return from `TAsynConnRequest` is equivalent to a `OK` return from `TConnRequest`). In the final case, when either `CONNECTING_1` or `CONNECTING_2` is returned, only the `tc_sd` element of the `tc` parameter has been updated; it reflects the transport-descriptor to be used to reference this connection. Note that the `data` parameter is still being referenced by *libtsap*(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 `TSelectMask`. In order to determine if the connection attempt was successful, the routine `TAsynRetryRequest` is called:

```
int     TAsynRetryRequest (sd, tc, td)
int     sd;
struct TSAPconnect *tc;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd:` the transport-descriptor;

`tc`: a pointer to a `TSAPconnect` 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.

Again, one of four values are returned: `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 order to make efficient use of the asychronous connection facility, it is necessary to understand a bit of its underlying mechanisms. From a temporal perspective, connection establishment consists of two parts:

1. establishing a reliable end-to-end connection; and,

2. exchanging connection establishment information.

In some cases, the underlying transport mechanisms accomplish both simultaneously (when the end-to-end connection is built, connection establishment information is also exchanged).

Thus, in order to to perform asynchronous connections effectively, use of `TAsynConnRequest` and `TAsynRetry` should reflect this two-step process:

1. Call `TAsynConnRequest` with the `async` parameter taking the value 1. If the return value was either `NOTOK` (the connection was not established), or `DONE` (the connection was established), then terminate this algorithm.

   Otherwise, a return value of `CONNECTING_1` or `CONNECTING_2` indicates that connection establishment process has begun. Remember this value.

2. At some point in the future, call `TSelectMask` to get an argument for `xselect`. Then call `xselect` checking to see if writing is permitted. Then call `xselect` checking for writing if the remembered value was `CONNECTING_1` and for reading if it was `CONNECTING_2`. If either call returns `NOTOK`, then a catastrophic error has occurred.

   Repeat this step as often as necessary until `xselect` says that reading or writing as required is permitted.

3. Call `TAsynRetry`. If the return value was either `NOTOK` (the connection was not established), or `DONE` (the connection was established), then terminate this algorithm.

   Otherwise, a return value of `CONNECTING_1` or `CONNECTING_2` indicates that connection establishment is *still* in progress. Remember this value and go back to the previous step.

Although this seems complicated, implementation of these rules is actually straight-forward. In most cases, your code will do some work unrelated to the connection

Note that this procedure is equally applicable to the higher-layers (session, presentation, and association control) which also provide asynchronous connection facilities. For example, at the application layer, the routines `AcAsynAssocRequest` and `AcAsynRetryRequest` would be used.

4.3 Data Transfer

Once the connection has been established, a transport-descriptor is used to reference the connection. This is usually the first parameter given to any of the remaining routines in the *libtsap*(3n) library. Further, the last parameter is usually a pointer to a `TSAPdisconnect` structure (as described on page 108). If a call to one of these routines fails, then the structure is updated. Otherwise, if the value of the `td_reason` element is associated with a fatal error, then the connection is closed. That is, a T-DISCONNECT.INDICATION event has occurred. The `DR_FATAL` macro can be used to determine this.

```
int     DR_FATAL (r)
int     r;
```

For protocol purists, the `DR_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     DR_OFFICIAL (r)
int     r;
```

4.3.1 Sending Data

There are three routines that may be used to send data. A call to the `TDataRequest` routine is equivalent to a T-DATA.REQUEST action on the part of the user.

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

The parameters to this procedure are:

`sd:` the transport-descriptor;

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

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

If the call to `TDataRequest` is successful, then the data has been queued for sending. Otherwise, the `td` parameter indicates the reason for failure.

The `TWriteRequest` routine is similar in nature to the `TDataRequest` routine, but uses a different set of parameters. The invocation is:

```
int     TWriteRequest (sd, uv, td)
int     sd;
struct udvec *uv;
struct TSAPdisconnect  *td;
```

While the parameters are:

`sd:` the transport-descriptor;

`uv:` the data to be written, described in a null-terminated array of scatter/gather elements:

```
struct udvec {
    caddr_t uv_base;
    int     uv_len;
};
```

The elements of the structure are:

`uv_base`: the base of an element; and,

`uv_cc`: the length of an element.

and,

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

If the call to `TWriteRequest` is successful, then the data has been queued for sending. Otherwise, the `td` parameter indicates the reason for failure.

A call to the `TExpdRequest` routine is equivalent to a T-EXPEDITED-DATA.REQUEST action on the part of the user.

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

The parameters to this procedure are:

`sd`: the transport-descriptor;

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

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

If the call to `TExpdRequest` is successful, then the data has been queued for expedited sending. Otherwise, the `td` parameter indicates the reason for failure.

4.3.2 Receiving Data

There is one routine that is used to read data, `TReadRequest`, a call to which is equivalent to waiting for a T-DATA.INDICATION or T-EXPEDITED-DATA.INDICATION event.

```
int     TReadRequest (sd, tx, secs, td)
int     sd;
struct TSAPdata *tx;
int     secs;
struct TSAPdisconnect  *td;
```

The parameters to this procedure are:

`sd:` the transport-descriptor;

`tx:` a pointer to the `TSAPdata` 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,

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

If the call to `TReadRequest` is successful, then the data has been read into the `tx` parameter.

```
struct TSAPdata {
    int     tx_expedited;

    int     tx_cc;
    struct qbuf tx_qbuf;
};
```

The elements of a `TSAPdata` structure are:

`tx_expedited:` whether the data was received via expedited transfer (i.e., an T-EXPEDITED-DATA.INDICATION event occurred);

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

`tx_qbuf:` the data that was read, in a buffer-queue form.

```
struct qbuf {
    struct qbuf *qb_forw;
    struct qbuf *qb_back;
```